Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Новосибирский государственный технический университет»

Кафедра прикладной математики

Курсовой проект по курсу

**«Уравнения математической физики»**

Группа ПМ-24

Студент ГЕРАСИМЕНКО ВАДИМ

Новосибирск

2025

# 1. Условие задачи

МКЭ для двумерной начально-краевой задачи для гиперболического уравнения в декартовой системе координат, схема Кранка-Николсона по времени. Базисные функции билинейные на прямоугольниках. Краевые условия всех типов. Коэффициент диффузии $\lambda$ разложить по биквадратичным базисным функциям. Матрицу СЛАУ генерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

# 2. Постановка задачи

Решаемое уравнение в общем виде:

$$\chi \frac{\partial^2 u}{\partial t^2} + \sigma \frac{\partial u}{\partial t} - div(\lambda \, grad\, u) = f \ ,$$

область интегрирования $\Omega$ с границей $S = S_1 \cup S_2 \cup S_3$ и краевыми условиями:

$$u|_{S_1} = u_g \ ,$$

$$\lambda \frac{\partial u}{\partial n}|_{S_2} = \theta \ ,$$

$$\lambda \frac{\partial u}{\partial n}|_{S_3} + \beta\left(u|_{S_3} - u_\beta\right) = 0 \ .$$

начальные условия:

$$u|_{t=t_0} = u^0 \ ,$$

$$u|_{t=t_1} = u^1$$

Дифференциальное уравнение для двумерной гиперболической начально-краевой задачи в декартовой системе координат:

$$\chi \frac{\partial^2 u}{\partial t^2} + \sigma \frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left(\lambda \frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(\lambda \frac{\partial u}{\partial y}\right) = f \ .$$

# 3. Теоретическая часть

## 3.1 Дискретизация по времени

При построении дискретного аналога для данной начально-краевой задачи будем полагать, что ось времени t разбита на так называемые временные слои значениями $t_j$ , j = 1…J, а значения искомой функции $u$ и параметров $\lambda$ , $\sigma$, $\chi$ и $f$ на j-м временном слое (т. е. при t = $t_j$) будем обозначать соответственно через $u^j$, $\lambda^j$ ,$\sigma^j$, $\chi^j$ и $f^j$, которые уже не зависят от времени t, но остаются функциями пространственных координат: $u^j = u^j(x,y) = u(x,y,\ t_j)$ , $\lambda^j = \lambda^j(x,y) = \lambda(x,y,\ t_j)$ и т. д.

Если считать, что параметры дифференциального уравнения не зависят от времени, а правая часть $f$ – зависит, то схема Кранка-Николсона будет иметь следующий вид:

$$\chi\frac{u^j-2u^{j-1}+u^{j-2}}{\Delta t^2}+\sigma\frac{u^j-u^{j-2}}{2\Delta t}-div\left(\lambda\, grad\,\frac{u^j+u^{j-2}}{2}\right)=\frac{f^j+f^{j-2}}{2}$$

где $u^j$ является значением искомой функции $u$ на временном слое $t=t_j$ , $\Delta t = t^j-t^{j-1}$.

Представим искомое решение $u$ на интервале $(t_{j-2},\ t_j)$ в следующем виде:

$$u(x,y,t)\approx u^{j-2}(x,y)\eta_2^j(t)+u^{j-1}(x,y)\eta_1^j(t)+u^j(x,y)\eta_0^j(t)\,.$$

В аппроксимации функции пространственных координат $u^{j-2}$, $u^{j-1}$ и $u^j$ являются значениями искомой функции $u$ при $t=t_{j-2}$, $t=t_{j-1}$ и $t=t_j$ соответственно. Зависящие только от времени функции $\eta_v^j(t)$ являются квадратичными полиномами $t$, причём $\eta_2^j(t)$ равна единице при $t=t_{j-2}$ и нулю при $t=t_{j-1}$ и $t=t_j$, функция $\eta_1^j(t)$ равна единице при $t=t_{j-1}$ и нулю при $t=t_{j-2}$ и $t=t_j$, а функция $\eta_0^j(t)$ равна единице при $t=t_j$ и нулю при $t=t_{j-2}$ и $t=t_{j-1}$. Таким образом, указанное соотношение определяет аппроксимацию функции $u$ по времени как квадратичный интерполянт её значений на временных слоях $t=t_{j-2}, t=t_{j-1}$ и $t=t_j$.

Очевидно, что функции $\eta_2^j(t)$, $\eta_1^j(t)$ и $\eta_0^j(t)$ – это базисные квадратичные полиномы Лагранжа (с двумя корнями из набора значений времён $t=t_{j-2}, t=t_{j-1}$ и $t=t_j$), которые могут быть записаны в виде

$$\eta_2^j(t)=\frac{1}{\Delta \mathrm{t}_1\Delta \mathrm{t}}(t-t_{j-1})(t-t_j)\,,$$

$$\eta_1^j(t)=-\frac{1}{\Delta \mathrm{t}_1\Delta \mathrm{t}_0}(t-t_{j-2})(t-t_j)\,,$$

$$\eta_2^j(t)=\frac{1}{\Delta \mathrm{t}\Delta \mathrm{t}_0}(t-t_{j-2})(t-t_1)\,.$$

где

$$\Delta \mathrm{t}=t_j-t_{j-2}\,,\qquad \Delta \mathrm{t}_1=t_{j-1}-t_{j-2}\,,\qquad \Delta \mathrm{t}_0=t_j-t_{j-1}\,.$$

Вычислим значения первой и второй производных этих полиномов при $t=t_{j-2}$ и $t=t_j$ :

$$\left.\frac{d\eta_2^j(t)}{dt}\right|_{t=t_j} = -\frac{\Delta t_0}{\Delta t_1 \Delta t}; \quad \left.\frac{d\eta_2^j(t)}{dt}\right|_{t=t_{j-2}} = -\frac{\Delta t + \Delta t_1}{\Delta t_1 \Delta t};$$

$$\left.\frac{d\eta_1^j(t)}{dt}\right|_{t=t_j} = -\frac{\Delta t}{\Delta t_1 \Delta t_0}; \quad \left.\frac{d\eta_2^j(t)}{dt}\right|_{t=t_{j-2}} = \frac{\Delta t}{\Delta t_1 \Delta t_0};$$

$$\left.\frac{d\eta_0^j(t)}{dt}\right|_{t=t_j} = \frac{\Delta t + \Delta t_0}{\Delta t \Delta t_0}; \quad \left.\frac{d\eta_2^j(t)}{dt}\right|_{t=t_{j-2}} = -\frac{\Delta t_1}{\Delta t \Delta t_0};$$

$$\left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_j} = \frac{2}{\Delta t_1 \Delta t}; \quad \left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_{j-2}} = \frac{2}{\Delta t_1 \Delta t};$$

$$\left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_j} = \frac{2}{\Delta t_1 \Delta t}; \quad \left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_{j-2}} = \frac{2}{\Delta t_1 \Delta t};$$

$$\left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_j} = \frac{2}{\Delta t_1 \Delta t}; \quad \left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_{j-2}} = \frac{2}{\Delta t_1 \Delta t}.$$

В итоге, первая и вторая производные по времени аппроксимируются как

$$\frac{\partial u}{\partial t} \approx -\frac{\Delta t_0 + \Delta t + \Delta t_1}{2\Delta t_1 \Delta t} u^{j-2} + \frac{-\Delta t + \Delta t}{2\Delta t_1 \Delta t_0} u^{j-1} + \frac{\Delta t + \Delta t_0 - \Delta t_1}{2\Delta t \Delta t_0} u^j = \frac{u^j - u^{j-2}}{\Delta t}$$

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{2}{\Delta t \Delta t_1} u^{j-2} - \frac{2}{\Delta t_1 \Delta t_0} u^{j-1} + \frac{2}{\Delta t \Delta t_0} u^j .$$

Тогда аппроксимация дифференциального уравнения по времени может быть записана в виде

$$\chi\left(\frac{2}{\Delta t \Delta t_1} u^{j-2} - \frac{2}{\Delta t_1 \Delta t_0} u^{j-1} + \frac{2}{\Delta t \Delta t_0} u^j\right) + \sigma\left(\frac{u^j - u^{j-2}}{\Delta t}\right)$$
$$-div\left(\lambda\, grad\, \frac{u^j + u^{j-2}}{2}\right) = \frac{f^j + f^{j-2}}{2}.$$

## 3.2 Вариационная постановка в форме уравнения Галеркина

Невязка $R(u)$ данного уравнения примет следующий вид:

$$R(u) = \chi\left(\frac{2}{\Delta t \Delta t_1} u^{j-2} - \frac{2}{\Delta t_1 \Delta t_0} u^{j-1} + \frac{2}{\Delta t \Delta t_0} u^j\right) + \sigma\left(\frac{u^j - u^{j-2}}{\Delta t}\right)$$
$$- div\left(\lambda\, grad\, \frac{u^j + u^{j-2}}{2}\right) - \frac{f^j + f^{j-2}}{2}.$$

Потребуем, чтобы эта невязка была ортогональна (в смысле скалярного произведения пространства $L_2(\Omega) \equiv H^0$) некоторому пространству $\Phi$ функций $v$, которое будем называть *пространством пробных функций*:

$$\int_{\Omega}\Big(\chi\Big(\tfrac{2}{\Delta t\Delta t_1}u^{j-2}-\tfrac{2}{\Delta t_1\Delta t_0}u^{j-1}+\tfrac{2}{\Delta t\Delta t_0}u^{j}\Big)+\ +\sigma\Big(\tfrac{u^j-u^{j-2}}{\Delta t}\Big)-div\Big(\lambda\,grad\ \tfrac{u^j+u^{j-2}}{2}\Big)-$$
$$-\tfrac{f^j+f^{j-2}}{2}\Big)\,v\,d\,\Omega=0\,,\forall v\in\Phi\,.$$

Воспользуемся формулой Грина и преобразуем интегралы по границам $S_2$ и $S_3$, воспользовавшись краевыми условиями:

$$\frac{1}{2}\int_{\Omega}\lambda^j\,grad(u^j)\,grad(v)\,d\Omega+\frac{1}{2}\int_{\Omega}\lambda^j grad(u^{j-2})\,grad(v)\,d\Omega+\frac{1}{2}\int_{S_3}\beta^j u^j v dS+$$
$$+\ \frac{1}{2}\int_{S_3}\beta^{j-2}u^{j-2}vdS\ +\frac{1}{\Delta t}\int_{\Omega}\sigma^j u^j vd\Omega\ -\frac{1}{\Delta t}\int_{\Omega}\sigma^j u^{j-2}vd\Omega\ +$$
$$+\frac{2}{\Delta t\Delta t_0}\int_{\Omega}\chi^j u^j vd\Omega\ -\frac{2}{\Delta t_1\Delta t_0}\int_{\Omega}\chi^j u^{j-1}vd\Omega\ +\frac{2}{\Delta t\Delta t_1}\int_{\Omega}\chi^j u^{j-2}vd\Omega\ =$$
$$=\tfrac{1}{2}\int_{\Omega}f^j vd\Omega\ -\tfrac{1}{2}\int_{\Omega}f^{j-2}vd\Omega\ +\tfrac{1}{2}\int_{S_2}\theta^j vdS\ +\tfrac{1}{2}\int_{S_2}\theta^{j-2}vdS+$$
$$+\tfrac{1}{2}\int_{S_3}\beta^j {u_\beta}^j vdS+\tfrac{1}{2}\int_{S_3}\beta^{j-2}{u_\beta}^{j-2}vdS\,,\qquad\qquad \forall v\in\Phi\,.$$

где $S=S_1\cup S_2\cup S_3$ $-$ граница $\Omega$.

В качестве $\Phi$ выберем $H_0^1-$ пространство пробных функций $v_0\in H^1$, которые на границе $S_1$ удовлетворяют нулевым первым краевым условиям, и при этом будем считать, что $u\in H_g^1$, где $H_g^1$ – множество функций, имеющих с квадратом первым производные и удовлетворяющих только первым краевым условиям на границе $S_1$.

Учитывая, что $v_0|_{S_1}=0$ , получим итоговое уравнение:

$$\frac{1}{2}\int_{\Omega}\lambda^j\,grad(u^j)\,grad(v_0)\,d\Omega+\frac{1}{2}\int_{\Omega}\lambda^j grad(u^{j-2})\,grad(v_0)\,d\Omega+\frac{1}{2}\int_{S_3}\beta^j u^j v_0 dS+$$
$$+\ \frac{1}{2}\int_{S_3}\beta^{j-2}u^{j-2}v_0dS\ +\frac{1}{\Delta t}\int_{\Omega}\sigma^j u^j v_0d\Omega\ -\frac{1}{\Delta t}\int_{\Omega}\sigma^j u^{j-2}v_0d\Omega\ +$$
$$+\ \frac{2}{\Delta t\Delta t_0}\int_{\Omega}\chi^j u^j v_0d\Omega\ -\frac{2}{\Delta t_1\Delta t_0}\int_{\Omega}\chi^j u^{j-1}v_0d\Omega\ +\frac{2}{\Delta t\Delta t_1}\int_{\Omega}\chi^j u^{j-2}v_0d\Omega\ =$$
$$=\tfrac{1}{2}\int_{\Omega}f^j v_0d\Omega\ -\tfrac{1}{2}\int_{\Omega}f^{j-2}v_0d\Omega\ +\tfrac{1}{2}\int_{S_2}\theta^j v_0dS\ +\tfrac{1}{2}\int_{S_2}\theta^{j-2}v_0dS+$$
$$+\tfrac{1}{2}\int_{S_3}\beta^j {u_\beta}^j v_0dS+\tfrac{1}{2}\int_{S_3}\beta^{j-2}{u_\beta}^{j-2}v_0dS\,,\qquad\qquad \forall v_0\in H_0^1\,.$$

## 3.3 Конечноэлементная дискретизация и переход к локальным матрицам

При построении конечноэлементных аппроксимаций по методу Галеркина пространства $H_g^1$ и $H_0^1$ заменяются конечномерными пространстами $V_g^h$ и $V_0^h$. При этом чаще всего в МКЭ функции из этих пространств являются элементами одного и того же конечномерного пространства $V^h$, которое мы будем определять как линейное пространство, натянутое на базисные функции $\psi_i$, $i = 1 \dots n$.

Как правило, функции $\psi_i$ являются финитными кусочно-полиномиальными функциями, а приближённое решение $u^h \in V_g^h$ является линейной комбинацией таких функций.

Для получения аппроксимации уравнения Галеркина на конечномерных пространствах $V_g^h$ и $V_0^h$ заменим в полученном уравнении функцию $u \in H_g^1$ аппроксимирующей её функцией $u^h \in V_g^h$, а функцию $v_0 \in H_0^1$ – функцией $v_0^h \in V_0^h$:

$$\frac{1}{2}\int_\Omega \lambda^j\, grad(u^j)\, grad(v_0^h)\, d\Omega + \frac{1}{2}\int_\Omega \lambda^j grad(u^{j-2})\, grad(v_0^h)\, d\Omega + \frac{1}{2}\int_{S_3} \beta^j u^j v_0^h dS +$$
$$+\frac{1}{2}\int_{S_3} \beta^{j-2} u^{j-2} v_0^h dS + \frac{1}{\Delta t}\int_\Omega \sigma^j u^j v_0^h d\Omega - \frac{1}{\Delta t}\int_\Omega \sigma^j u^{j-2} v_0^h d\Omega +$$
$$+\frac{2}{\Delta t \Delta t_0}\int_\Omega \chi^j u^j v_0^h d\Omega - \frac{2}{\Delta t_1 \Delta t_0}\int_\Omega \chi^j u^{j-1} v_0^h d\Omega + \frac{2}{\Delta t \Delta t_1}\int_\Omega \chi^j u^{j-2} v_0^h d\Omega =$$
$$= \frac{1}{2}\int_\Omega f^j v_0^h d\Omega - \frac{1}{2}\int_\Omega f^{j-2} v_0^h d\Omega + \frac{1}{2}\int_{S_2} \theta^j v_0^h dS + \frac{1}{2}\int_{S_2} \theta^{j-2} v_0^h dS +$$
$$+\frac{1}{2}\int_{S_3} \beta^j {u_\beta}^j v_0^h dS + \frac{1}{2}\int_{S_3} \beta^{j-2} {u_\beta}^{j-2} v_0^h dS, \qquad \forall v_0^h \in V_0^h.$$

Так как любая функция $v_0^h \in V_0^h$ может быть представлена в виде линейной комбинации:

$$v_0^h = \sum_{i \in N_0} q_i^v \psi_i,$$

вариационное уравнение эквивалентно следующей системе уравнений:

$$\frac{1}{2}\int_\Omega \lambda^j\, grad(u^{hj})\, grad(\psi_i)\, d\Omega + \frac{1}{2}\int_\Omega \lambda^j grad(u^{hj-2})\, grad(\psi_i)\, d\Omega + \frac{1}{2}\int_{S_3} \beta^j u^{hj} \psi_i dS +$$
$$+\frac{1}{2}\int_{S_3} \beta^{j-2} u^{hj-2} \psi_i dS + \frac{1}{\Delta t}\int_\Omega \sigma^j u^{hj} \psi_i d\Omega - \frac{1}{\Delta t}\int_\Omega \sigma^j u^{hj-2} \psi_i d\Omega +$$
$$+\frac{2}{\Delta t \Delta t_0}\int_\Omega \chi^j u^{hj} \psi_i d\Omega - \frac{2}{\Delta t_1 \Delta t_0}\int_\Omega \chi^j u^{hj-1} \psi_i d\Omega + \frac{2}{\Delta t \Delta t_1}\int_\Omega \chi^j u^{hj-2} \psi_i d\Omega =$$
$$= \frac{1}{2}\int_\Omega f^j \psi_i d\Omega - \frac{1}{2}\int_\Omega f^{j-2} \psi_i d\Omega + \frac{1}{2}\int_{S_2} \theta^j \psi_i dS + \frac{1}{2}\int_{S_2} \theta^{j-2} \psi_i dS +$$
$$+\frac{1}{2}\int_{S_3} \beta^j {u_\beta}^j \psi_i dS + \frac{1}{2}\int_{S_3} \beta^{j-2} {u_\beta}^{j-2} \psi_i dS, \qquad i \in N_0.$$

Таким образом, МКЭ-решение $u^h$ удовлетворяет полученной системе уравнений. Поскольку $u^h \in V_g^h$, оно может быть представлено в виде линейной комбинации базисных функций пространства $V^h$:

$$u^h = \sum_{k=1}^{n} q_k \psi_k.$$

Подставляя данное выражение в ранее полученную систему уравнений, получаем решение для компонент $q_k$ вектора весов $\boldsymbol{q} = (q_1, \dots, q_n)^T$ с индексами $k \in N_0$:

$$\sum_{k=1}^{n} \left( \frac{1}{2}\int_\Omega \lambda^j\, grad(\psi_k^j)\, grad(\psi_i)\, d\Omega + \frac{1}{\Delta t}\int_\Omega \sigma^j \psi_k^j \psi_i d\Omega + \frac{2}{\Delta t \Delta t_0}\int_\Omega \chi^j \psi_k^j \psi_i d\Omega + \right.$$
$$\left. + \frac{1}{2}\int_{S_3} \beta^j \psi_k^j \psi_i dS \right) q_k^j = \frac{1}{2}\int_\Omega f^j \psi_i d\Omega - \frac{1}{2}\int_\Omega f^{j-2}\psi_i d\Omega + \frac{1}{2}\int_{S_2} \theta^j \psi_i dS + \frac{1}{2}\int_{S_2} \theta^{j-2}\psi_i dS +$$
$$+ \frac{1}{2}\int_{S_3} \beta^j u_\beta{}^j \psi_i dS + \frac{1}{2}\int_{S_3} \beta^{j-2} u_\beta{}^{j-2} \psi_i dS +$$
$$+ \sum_{k=1}^{n} \left( \frac{2}{\Delta t_1 \Delta t_0}\int_\Omega \chi^j \psi_k^{j-1} \psi_i d\Omega \right) q_k^{j-1} + \sum_{k=1}^{n} \left( -\frac{1}{2}\int_\Omega \lambda^j\, grad(\psi_k^{j-2})\, grad(\psi_i)\, d\Omega - \right.$$
$$\left. - \frac{1}{2}\int_{S_3} \beta^j \psi_k^{j-2}\psi_i dS + \frac{1}{\Delta t}\int_\Omega \sigma^j \psi_k^{j-2}\psi_i d\Omega - \frac{2}{\Delta t \Delta t_1}\int_\Omega \chi^j \psi_k^{j-2}\psi_i d\Omega \right) q_k^{j-2}, \qquad i, k \in N_0.$$

Сборка глобальных матрицы и вектора правой части выполняется из локальных матриц и векторов конечных элементов. При этом локальная матрица представляет собой сумму двух матриц: матрицы жесткости и матрицы массы, где элементы матрицы жесткости определяются как

$$\hat{G}_{ik} = \int_{\Omega_l} \lambda^j \left( \frac{\partial \hat{\psi}_k}{\partial x}\frac{\partial \hat{\psi}_i}{\partial x} + \frac{\partial \hat{\psi}_k}{\partial y}\frac{\partial \hat{\psi}_i}{\partial y} \right) d\Omega\,,$$

элементы матрицы масс:

$$\hat{M}^\sigma{}_{ik} = \int_{\Omega_l} \sigma^j \hat{\psi}_k \hat{\psi}_i d\Omega\,, \qquad \hat{M}^\chi{}_{ik} = \int_{\Omega_l} \chi^j \hat{\psi}_k \hat{\psi}_i d\Omega\,,$$

Обозначим вклады в глобальную матрицу от краевых условий третьего рода как $M^{S_3}$, и учтём, что запись $\boldsymbol{b}^j$ означает, что этот вектор сформирован по значениям функции $f$ и краевых условий на $j$-м временном слое.

Если считать векторы $q^{j-1}$ и $q^{j-2}$ известными (их компоненты являются весами разложения по базисным функциям решения на ($j$ – *1*)-м и ($j$ – *2*)-м временных слоях), то получим СЛАУ вида $\boldsymbol{A}q^j = \boldsymbol{d}$ , где:

$$\boldsymbol{A} = \frac{2}{\Delta t \Delta t_0} M^\chi + \frac{1}{\Delta t} M^\sigma + \frac{1}{2} G + \frac{1}{2} M^{S_3},$$

$$\boldsymbol{d} = \frac{1}{2} b^j + \frac{1}{2} b^{j-2} + \frac{2}{\Delta t_1 \Delta t_0} M^\chi q^{j-1} - \frac{2}{\Delta t \Delta t_1} M^\chi q^{j-2} + \frac{1}{\Delta t} M^\sigma q^{j-2} - \frac{1}{2} G q^{j-2} - \frac{1}{2} M^{S_3} q^{j-2}\,.$$

В конечном итоге СЛАУ для неравномерной сетки по времени примет вид:

$$\boldsymbol{A} = \frac{2}{(t^j - t^{j-2})(t^j - t^{j-1})} M^\chi + \frac{1}{t^j - t^{j-2}} M^\sigma + \frac{1}{2} G + \frac{1}{2} M^{S_3},$$

$$\boldsymbol{d} = \frac{1}{2} b^j + \frac{1}{2} b^{j-2} + \frac{2}{(t^{j-1} - t^{j-2})(t^j - t^{j-1})} M^\chi q^{j-1} - \frac{2}{(t^j - t^{j-2})(t^{j-1} - t^{j-2})} M^\chi q^{j-2} +$$

$$+\frac{1}{t^j - t^{j-2}} M^{\sigma} q^{j-2} - \frac{1}{2} G q^{j-2} - \frac{1}{2} M^{S_3} q^{j-2}.$$

## 3.4 Базисные функции

Для реализации МКЭ для данной задачи необходимо разбить область $\Omega$ на прямоугольные конечные элементы $\Omega_{ps} = [x_p,\ x_{p+1}] \times [y_s,\ y_{s+1}]$ и определить построение билинейных базисных функций.

Зададим по две одномерные линейные функции на каждом из отрезков:

$$X_1(x) = \frac{x_{p+1} - x}{h_x},\ X_2(x) = \frac{x - x_p}{h_x},\ h_x = x_{p+1} - x_p,$$

$$Y_1(y) = \frac{y_{s+1} - y}{h_y},\ Y_2(y) = \frac{y - y_s}{h_y},\ h_y = y_{s+1} - y_s,$$

Локальные базисные функции на конечном элементе $\Omega_{ps} = \left[x_p,\ x_{p+1}\right] \times [y_s,\ y_{s+1}]$ представляются в виде произведения функций:

$$\hat{\psi}_1(x,y) = X_1(x)Y_1(y), \quad \hat{\psi}_2(x,y) = X_2(x)Y_1(y),$$

$$\hat{\psi}_3(x,y) = X_1(x)Y_2(y), \quad \hat{\psi}_4(x,y) = X_2(x)Y_2(y).$$

Если параметр $\lambda$ на конечном элементе $\Omega_{ps}$ заменить его осредненным значением $\bar{\lambda}$ и учесть, что

$$\int_{x_p}^{x_p+h_x} \left(\frac{dX_1}{dx}\right)^2 dx = \frac{1}{h_x},\ \int_{x_p}^{x_p+h_x} \frac{dX_1}{dx}\frac{dX_2}{dx} dx = -\frac{1}{h_x},$$

$$\int_{x_p}^{x_p+h_x} \left(\frac{dX_2}{dx}\right)^2 dx = \frac{1}{h_x},\ \int_{x_p}^{x_p+h_x} (X_1)^2 dx = \frac{h_x}{3},$$

$$\int_{x_p}^{x_p+h_x} X_1 X_2 dx = \frac{h_x}{6},\ \int_{x_p}^{x_p+h_x} (X_2)^2 dx = \frac{h_x}{3}.$$

(аналогичный вид будут иметь и интегралы от произведений функций $Y_v(y)$ и их производных), а также заменить $\sigma$ на $\bar{\sigma}$ и $\chi$ на $\bar{\chi}$, то, подставляя в ранее полученные формулы компонент матриц жёсткости и массы, преобразовав, получим:

$$\hat{G} = \frac{\bar{\lambda}}{6}\frac{h_y}{h_x}\begin{pmatrix} 2 & -2 & 1 & -1 \\ -2 & 2 & -1 & 1 \\ 1 & -1 & 2 & -2 \\ -1 & 1 & -2 & 2 \end{pmatrix} + \frac{\bar{\lambda}}{6}\frac{h_x}{h_y}\begin{pmatrix} 2 & 1 & -2 & -1 \\ 1 & 2 & -1 & -2 \\ -2 & -1 & 2 & 1 \\ -1 & -2 & 1 & 2 \end{pmatrix},$$

$$\widehat{M}^{\sigma} = \bar{\sigma}\hat{C} = \bar{\sigma}\frac{h_x h_y}{36}\begin{pmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{pmatrix}, \widehat{M}^{\chi} = \bar{\chi}\hat{C} = \bar{\chi}\frac{h_x h_y}{36}\begin{pmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{pmatrix}.$$

Если представить правую часть $f$ решаемого уравнения в виде билинейного интерполянта $\sum_{v=1}^{4} \hat{f}_v \hat{\psi}_v$, то локальный вектор $\widehat{\boldsymbol{b}}$ в этом случае легко вычисляется через матрицу $\hat{C}$, фактически являющуюся матрицей массы при $\bar{\sigma} \equiv 1$:

$$\widehat{\boldsymbol{b}} = \widehat{\boldsymbol{C}} \cdot \hat{\boldsymbol{f}} = \frac{h_x h_y}{36}\begin{pmatrix} 4\hat{f}_1 + 2\hat{f}_2 + 2\hat{f}_3 + \hat{f}_4 \\ 2\hat{f}_1 + 4\hat{f}_2 + \hat{f}_3 + 2\hat{f}_4 \\ 2\hat{f}_1 + \hat{f}_2 + 4\hat{f}_3 + 2\hat{f}_4 \\ \hat{f}_1 + 2\hat{f}_2 + 2\hat{f}_3 + 4\hat{f}_4 \end{pmatrix}.$$

# 4. Занесение данных в программу

### 4.1 Способ задания расчётной области

Для задания расчётной области служит файл “AreaDescription.txt”, задающий расчетную область в виде подобластей.

В первой строке файла одно целое число n – количество вертикальных границ подобластей, во второй строке n чисел – координаты по оси X этих границ, так определяется массив xValues в классе Area. Следующие две строки аналогичны для горизонтальных границ и определяют массив yValues того же класса.

В пятой строке одно целое число k – количество подобластей. Следующие k строк описывают подобласти. Каждая представляется набором из пяти целых чисел. Первое обозначает индекс подобласти/номер формул, определяющих параметры дифференциального уравнения в рассматриваемой подобласти, второе и третье – индексы левой и правой вертикальных границ соответственно в массиве xValues, четвертое и пятое – индексы нижней и верхней горизонтальных границ соотвественно из массива yValues.

### 4.2 Способ задания краевых условий

Для задания краевых условий служит файл “BoundariesDescription.txt”. Каждый фрагмент границ $S_1$, $S_2$ и $S_3$ может быть описан шестью целыми числами:

- тип краевого условия (1 означает принадлежность границе $S_1$, 2 – границе $S_2$, 3 – границе $S_3$);
- индекс формулы, задающей параметр краевого условия на рассматриваемом фрагменте границы);

- индекс элемента массива Area.xValues, с которого рассматриваемый фрагмент начинается по оси x;
- индекс элемента массива Area.xValues, которым фрагмент рассматриваемый фрагмент заканчивается по оси x;
- индекс элемента массива Area.yValues, с которого рассматриваемый фрагмент начинается по оси y;
- индекс элемента массива Area.yValues, которым фрагмент рассматриваемый фрагмент заканчивается по оси y;

### 4.3 Способ задания сетки по пространству

Двумерная регулярная прямоугольная сетка хранится в виде двух массивов xValues и yValues класса Grid. Эти массивы строятся на основе одноименных массивов из класса Area, разбиваемых на более мелкие отрезки на основе информации из файла “IntervalsDescription.txt”.

Первая строка файла “IntervalsDescription.txt” содержит n пар чисел, где n = количество интервалов в массиве Area.xValues. Первое число в каждой паре (целое) определяет количество интервалов, на которое должен быть разбит каждый интервал. Второе число в паре (вещественное) – коэффициент изменения длины интервалов при разбиении каждого подынтервала. Аналогично, вторая строка содержит m пар, где m = количество интервалов в массиве Area.yValues. Пары чисел во второй строке по смыслу соответствуют парам чисел из первой строки.

### 4.4 Способ задания сетки по времени

Сетка по времени описывается отдельным программным классом **TimeLayers**, принимающим в конструктор несколько аргументов, которые позволяют описать как равномерную, так и неравномерную сетку.

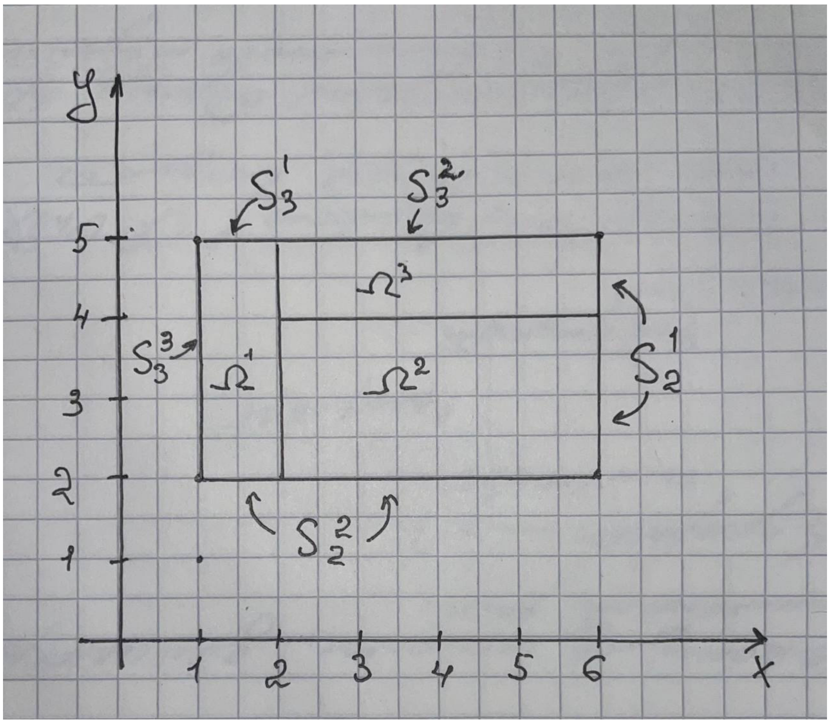
Для задания неравномерной сетки служит один конструктор принимающий массив вещественных чисел. Каждое число обозначает временной слой.

Для задания равномерной сетки служат два специализированных конструктора, а также способ задания в виде неравномерной сетки. Первый конструктор принимает в качестве аргументов два вещественных числа и одно целое: начальная секунда, конечная секунда и количество временных слоёв соответственно. Второй конструктор принимает одно вещественное число и одно целое число: количество секунд наблюдения и количество временных слоёв, на которое разбивается время наблюдения; данный конструктор вызывает первый конструктор, выбирая в качестве начала отсчёта нулевое время.

## 5. Тестирование программы

### 5.1 Порядок аппроксимации

Расчетная область:




Первая исследуемая функция:

$$u = xt^2,$$

$$\lambda = 1\,,\quad \chi = 1\,,\ \sigma = 0.1,$$

$$f = 2x + 0.2xt\,,$$

$$\lambda\frac{\partial u}{\partial n}|_{S_2^1} = t^2\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - xt^2)\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - xt^2)\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u - xt^2 + 2t^2)\right)|_{S_3^3} = 0\,,$$

$$u|_{t=t_0} = xt_0^2,\qquad \frac{\partial u}{\partial t}|_{t=t_0} = 2xt_0\,.$$

Равномерная сетка по времени: Неравномерная сетка по времени:

| Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ | Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ |
|---|---|---|---|
| 0.1 | 0.0000000000000000E-00 | 0.20 | 0.0000000000000000E-00 |
| 0.2 | 0.0000000000000000E-00 | 0.70 | 0.0000000000000000E-00 |

| | | | |
|---|---|---|---|
| 0.3 | 1.2384294633765546E-15 | 0.80 | 3.1086816377379200E-16 |
| 0.4 | 1.0670650354049723E-15 | 1.00 | 4.6610881143838320E-16 |
| 0.5 | 1.5842298504036879E-15 | 1.15 | 5.7427827344317400E-16 |
| 0.6 | 2.2003192366717890E-15 | 1.20 | 9.2904320075209470E-16 |
| 0.7 | 2.5448282484355430E-15 | 1.29 | 1.6061513564464037E-15 |
| 0.8 | 2.7167130050616600E-15 | 1.44 | 3.0006495696218103E-15 |
| 0.9 | 3.1260661734207464E-15 | 1.80 | 4.9725901658413326E-15 |
| 1.0 | 3.1060444830557713E-15 | 2.00 | 5.2492680648658550E-15 |

Ожидаемо, если исключить вычислительную погрешность, то погрешность отсутствует, так как порядок полинома по пространственным координатам не превышает порядка используемых базисных функций, а порядок полинома по времени соответствует порядку точности используемой временной схемы (полином второй степени).

Вторая исследуемая функция:

$$u = xt^3,$$

$$\lambda = 1\,,\quad \chi = 1\,,\ \sigma = 0.1,$$

$$f = 6xt + 0.3xt^2\,,$$

$$\lambda\frac{\partial u}{\partial n}|_{S_2^1} = t^3\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - xt^3)\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - xt^3)\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u - xt^3 + 2t^3)\right)|_{S_3^3} = 0\,,$$

$$u|_{t=t_0} = xt_0^3,\qquad \frac{\partial u}{\partial t}|_{t=t_0} = 3xt_0^2\,.$$

Равномерная сетка по времени: Неравномерная сетка по времени:

| Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ | Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ |
|---|---|---|---|
| 0.1 | 0.000000000000000E-00 | 0.20 | 0.0000000000000000E-00 |
| 0.2 | 0.000000000000000E-00 | 0.70 | 0.0000000000000000E-00 |
| 0.3 | 7.300142223777171E-04 | 0.80 | 2.1701905105454140E-02 |

| | | | |
|---|---|---|---|
| 0.4 | 9.153585726308182E-04 | 1.00 | 2.9280871871427937E-02 |
| 0.5 | 9.262402695747154E-04 | 1.15 | 2.8250390650953410E-02 |
| 0.6 | 8.809232460048631E-04 | 1.20 | 2.7642623592571306E-02 |
| 0.7 | 8.192401616764896E-04 | 1.29 | 2.6012535698748850E-02 |
| 0.8 | 7.556984624982692E-04 | 1.44 | 2.2590310667224360E-02 |
| 0.9 | 6.956931238639238E-04 | 1.80 | 1.2823492416893370E-02 |
| 1.0 | 6.410454528847320E-04 | 2.00 | 1.0707069544755839E-02 |

Возникает погрешность, так как порядок полинома по времени выше порядка точности используемой временной схемы. Тот факт, что для неравномерной сетки погрешность оказалась выше, объясним тем, что схема Кранка-Николсона при довольно резких изменениях по времени может вызывать высокую осциллирующую погрешность даже при небольших временных промежутках.

Для уменьшения осциллирующей погрешности в численном решении, полученном с использованием схемы Кранка-Николсона, применяется следующий способ: значения решения нужно выдавать не в узлах временной сетки $t_j$, а в серединах $t_{j+\frac{1}{2}} = \frac{t_j + t_{j+1}}{2}$ интервалов $(t_j, t_{j+1})$ в виде полусуммы значений на соответствующих временных слоях: $u\left(t_{j+\frac{1}{2}}\right) = \frac{u(t_j) + u(t_{j+1})}{2}$.

Третья исследуемая функция:

$$u = xt^4,$$

$$\lambda = 1\,,\quad \chi = 1\,,\ \sigma = 0.1,$$

$$f = 12xt^2 + 0.4xt^3\,,$$

$$\lambda\frac{\partial u}{\partial n}|_{S_2^1} = t^4\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - xt^4)\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - xt^4)\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u - xt^4 + 2t^4)\right)|_{S_3^3} = 0\,,$$

$$u|_{t=t_0} = xt_0^4,\qquad \frac{\partial u}{\partial t}|_{t=t_0} = 4xt_0^3\,.$$

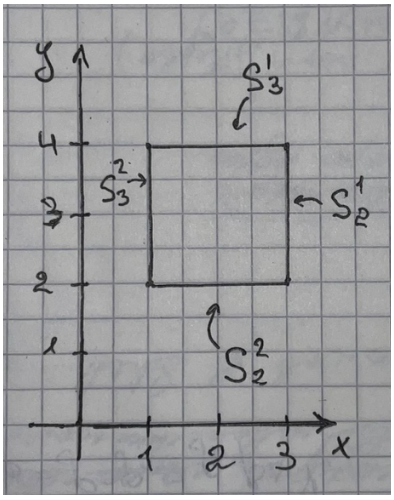
| Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ | Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 0.1 | 0.0000000000000000E-00 | 0.20 | 0.00000000000000000E-00 |
| 0.2 | 0.0000000000000000E-00 | 0.70 | 0.00000000000000000E-00 |
| 0.3 | 1.2361574165593753E-01 | 0.80 | 3.97678889366914800E-03 |
| 0.4 | 1.1655849725672987E-01 | 1.00 | 2.13856978716628480E-02 |
| 0.5 | 9.4607006334551820E-02 | 1.15 | 2.21068101291778850E-02 |
| 0.6 | 7.5183594460318790E-02 | 1.20 | 2.04884061312189080E-02 |
| 0.7 | 6.0093263586320896E-02 | 1.29 | 1.82575414703035730E-02 |
| 0.8 | 4.8635764465320190E-02 | 1.44 | 1.65931638821987700E-02 |
| 0.9 | 3.9908252249685555E-02 | 1.80 | 2.71692496750460050E-02 |
| 1.0 | 3.3187101892308314E-02 | 2.00 | 2.37713057569755000E-02 |

Аналогично предыдущему тесту, имеется погрешность, объяснимая доминированием порядка полинома по времени по отношению к порядку точности временной схемы.

### 5.3 Порядок сходимости

Расчётная область:




Для исследования возьмем не являющуюся полиномом функцию:

$$u = x\sin(t),$$

$$\lambda = 1\,,\quad \chi = 1\,,\ \sigma = 0.1,$$

$$f = -x\sin(t) + 0.1\cos(t),$$

$$\lambda\frac{\partial u}{\partial n}|_{S_2^1} = \sin(t)\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - x\sin(t))\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - x\sin(t))\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u - x\sin(t) + 2\sin(t))\right)|_{S_3^3} = 0\,,$$

$$u|_{t=t_0} = xsin(t_0), \qquad \frac{\partial u}{\partial t}|_{t=t_0} = xcos(t_0)\,.$$

10 временных слоёв: 20 временных слоёв:

| Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ | Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ |
|---|---|---|---|
| 0.1 | 0.0000000000000000E-00 | 0.1 | 0.0000000000000000E-00 |
| 0.2 | 0.0000000000000000E-00 | 0.2 | 9.1827938765736090E-07 |
| 0.3 | 1.6770923354561400E-05 | 0.3 | 5.4305780313342520E-06 |
| 0.4 | 4.8385591693540706E-05 | 0.4 | 1.3512050900507717E-05 |
| 0.5 | 9.4728073682502790E-05 | 0.5 | 2.5224573046058716E-05 |
| 0.6 | 1.5610758527812702E-04 | 0.6 | 4.0675219362992210E-05 |
| 0.7 | 2.3308299245009868E-04 | 0.7 | 6.0016044173838200E-05 |
| 0.8 | 3.2644604077783130E-04 | 0.8 | 8.3450884410976350E-05 |
| 0.9 | 4.3724788860311707E-04 | 0.9 | 1.1124573526633113E-04 |
| 1.0 | 5.6684786688189840E-04 | 1.0 | 1.4374250532332150E-04 |

40 временных слоёв:

| Время: | $\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ |
|---|---|
| 0.1 | 0.0000000000000000E-00 |
| 0.2 | 3.0362867768587846E-07 |
| 0.3 | 1.4571948504791152E-06 |
| 0.4 | 3.4914094387844960E-06 |
| 0.5 | 6.4290971913376400E-06 |

| | |
|---|---|
| 0.6 | 1.0299452528298306E-05 |
| 0.7 | 1.5141508926913208E-05 |
| 0.8 | 2.1006725910490774E-05 |
| 0.9 | 2.7961883087097045E-05 |
| 1.0 | 3.6092643172560960E-05 |

Отношение погрешностей:

| Время: | 10 слоёв / 20 слоёв | Время: | 20 слоёв / 40 слоёв |
|---|---|---|---|
| 0.1 | - | 0.1 | - |
| 0.2 | - | 0.2 | 3.02 |
| 0.3 | 3.09 | 0.3 | 3.72 |
| 0.4 | 3.58 | 0.4 | 3.87 |
| 0.5 | 3.75 | 0.5 | 3.92 |
| 0.6 | 3.84 | 0.6 | 3.95 |
| 0.7 | 3.88 | 0.7 | 3.96 |
| 0.8 | 3.91 | 0.8 | 3.97 |
| 0.9 | 3.93 | 0.9 | 3.98 |
| 1.0 | 3.94 | 1.0 | 3.98 |

Поскольку схема Кранка-Николсона обладает квадратичной скоростью сходимости, при дроблении сетки по времени в два раза погрешность должна была упасть в четыре раза. Результаты, полученные на практике, очень близки к теоретическим.

## 6. Текст программы

Программа написана на объектно-ориентированном языке программирования Java.

```
package ru.nstu.hyperbolicequation;

import ru.nstu.hyperbolicequation.slae.InitialTimeFunction;
import ru.nstu.hyperbolicequation.slae.Matrix;
import ru.nstu.hyperbolicequation.solutionarea.*;
import ru.nstu.hyperbolicequation.function.ThreeArityFunction;

import java.io.File;
import java.util.List;

public class App {
    public static void main(String[] args) {
        List<ThreeArityFunction<Double, Double, Double, Double>> lambdaFunctions = List.of(
                (x, y, t) -> 1.
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> gammaFunctions = List.of(
                (x, y, t) -> 0.
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> sigmaFunctions = List.of(
                (x, y, t) -> 0.1
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> chiFunctions = List.of(
                (x, y, t) -> 1.
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> rightFunctions = List.of(
                (x, y, t) -> -x * Math.sin(t) + 0.1 * x * Math.cos(t)
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> firstKind = List.of(
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> secondKind = List.of(
                (x, y, t) -> Math.sin(t),
                (x, y, t) -> 0.
        );

        List<ThreeArityFunction<Double, Double, Double, Double>> thirdKind = List.of(
                (x, y, t) -> x *  Math.sin(t),
                (x, y, t) -> x *  Math.sin(t) - 2 * Math.sin(t)
        );

        List<Double> betaCoefficients = List.of(1., 0.5);

        ThreeArityFunction<Double, Double, Double, Double> initTimeFunction = (x, y, t) -> x *
Math.sin(t);
        ThreeArityFunction<Double, Double, Double, Double> initTimeFunctionPartialDerivative =
(x, y, t) -> x * Math.cos(t);

        DescriptionFiles descriptionFiles = new DescriptionFiles(
                new File(directoryPath + "AreaDescription.txt"),
                new File(directoryPath + "BoundariesDescription.txt"),
                new File(directoryPath + "IntervalsDescription.txt")
        );

        InitialTimeFunction initialTimeFunction = new InitialTimeFunction(initTimeFunction,
initTimeFunctionPartialDerivative);
        TimeLayers timeLayers = new TimeLayers(0.1, 1, 55);

        HyperbolicEquation equation = new HyperbolicEquation(
```

```java
                lambdaFunctions, gammaFunctions, sigmaFunctions, chiFunctions,
                rightFunctions,
                firstKind, secondKind, thirdKind,
                betaCoefficients,
                initialTimeFunction,
                timeLayers,
                descriptionFiles
        );

        Matrix result = equation.solve();
        System.out.println(result);
    }
}
```

```java
package ru.nstu.hyperbolicequation.solutionarea;

import java.io.File;

public class DescriptionFiles {
    private final File areaDescription;
    private final File boundariesDescription;
    private final File intervalsDescription;

    public DescriptionFiles(File areaDescription, File boundariesDescription, File
intervalsDescription) {
        this.areaDescription = areaDescription;
        this.boundariesDescription = boundariesDescription;
        this.intervalsDescription = intervalsDescription;
    }

    public File getAreaDescription() {
        return areaDescription;
    }

    public File getBoundariesDescription() {
        return boundariesDescription;
    }

    public File getIntervalsDescription() {
        return intervalsDescription;
    }
}
```

```java
package ru.nstu.hyperbolicequation.solutionarea;

import java.util.Arrays;

public class TimeLayers {
    private final double[] layers;

    public TimeLayers(double startSeconds, double endSeconds, int layersCount) {
        if (startSeconds < 0 || endSeconds < 0) {
            throw new IllegalArgumentException("Seconds cannot be negative");
        }

        if (startSeconds >= endSeconds) {
            throw new IllegalArgumentException("Start seconds cannot be greater than end
seconds");
        }

        if (layersCount < 1) {
            throw new IllegalArgumentException("LayersCount cannot be less than 1");
        }

        layers = new double[layersCount];
        layers[0] = startSeconds;

        double secondsPerLayer = (endSeconds - startSeconds) / (layersCount - 1);
```

```java
        for (int i = 1; i < layersCount; i++) {
            layers[i] = startSeconds + secondsPerLayer * i;
        }
    }

    public TimeLayers(double seconds, int layersCount) {
        this(0.0, seconds, layersCount);
    }

    public TimeLayers(double[] layers) {
        this.layers = layers;
    }

    public double getTime(int layer) {
        return layers[layer];
    }

    public int countLayers() {
        return layers.length;
    }

    public double[] getLayers() {
        return layers.clone();
    }

    @Override
    public String toString() {
        return  "Time layers: " + Arrays.toString(getLayers());
    }
}

package ru.nstu.hyperbolicequation.solutionarea;

import ru.nstu.hyperbolicequation.function.ThreeArityFunction;
import ru.nstu.hyperbolicequation.slae.DenseMatrix;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class BoundaryConditions {
    Map<Integer, List<List<Integer>>> edges;

    List<ThreeArityFunction<Double, Double, Double, Double>> secondKind;
    List<ThreeArityFunction<Double, Double, Double, Double>> firstKind;
    List<ThreeArityFunction<Double, Double, Double, Double>> thirdKind;
    List<Double> beta;

    public static final int X_START_POSITION = 1;
    public static final int Y_START_POSITION = 3;

    public static final int FIRST_CONDITION_NUMBER = 1;
    public static final int SECOND_CONDITION_NUMBER = 2;
    public static final int THIRD_CONDITION_NUMBER = 3;

    public static final DenseMatrix LOCAL_MATRIX = new DenseMatrix(new double[][]{{2, 1}, {1,
2}});
    public static final double COMMON_LOCAL_MATRIX_DIVISOR = 6;

    public BoundaryConditions(File description,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
firstKind,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
secondKind,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
thirdKind,
                              List<Double> beta) {
        final int conditionsMaxCount = 3;
        final int propertiesCount = 5;

        try (Scanner scanner = new Scanner(description)) {
```

```java
            edges = new HashMap<>(conditionsMaxCount);

            for (int i = 1; i <= conditionsMaxCount; ++i) {
                edges.put(i, new ArrayList<>());
            }

            while (scanner.hasNext()) {
                int typeNumber = scanner.nextInt();
                List<List<Integer>> type = edges.get(typeNumber);
                List<Integer> edge = new ArrayList<>();

                for (int i = 0; i < propertiesCount; ++i) {
                    edge.add(scanner.nextInt());
                }

                type.add(edge);
                edges.put(typeNumber, type);
            }

            this.firstKind = firstKind;
            this.secondKind = secondKind;
            this.thirdKind = thirdKind;
            this.beta = beta;
        } catch (FileNotFoundException e) {
            System.out.println("Error reading file \"" + description.getName() + "\"" + e);
            System.exit(0);
        }
    }
}
```

```java
package ru.nstu.hyperbolicequation.solutionarea;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.Scanner;

public class Area {
    Subarea[] subareas;
    double[] xValues;
    double[] yValues;

    HyperbolicEquation equation;

    public Area(File description, HyperbolicEquation equation) {
        try (Scanner scanner = new Scanner(description)) {
            int xValuesCount = scanner.nextInt();
            xValues = new double[xValuesCount];

            for (int i = 0; i < xValuesCount; ++i) {
                xValues[i] = scanner.nextDouble();
            }

            int yValuesCount = scanner.nextInt();
            yValues = new double[yValuesCount];

            for (int i = 0; i < yValuesCount; ++i) {
                yValues[i] = scanner.nextDouble();
            }

            int subareasCount = scanner.nextInt();
            subareas = new Subarea[subareasCount];

            for (int i = 0; i < subareasCount; ++i) {
                int subareaIndex = scanner.nextInt();
                subareas[i] = new Subarea(this, subareaIndex,
                        scanner.nextInt(),
                        scanner.nextInt(),
                        scanner.nextInt(),
                        scanner.nextInt()
```

```java
                );
            }

            this.equation = equation;
        } catch (FileNotFoundException e) {
            System.out.println("Error reading file \""
                    + description.getName() + "\": " + e);
            System.exit(0);
        }
    }

    @Override
    public String toString() {
        return "Area X: " + Arrays.toString(xValues) + System.lineSeparator()
                + "Area Y: " + Arrays.toString(yValues);
    }
}


package ru.nstu.hyperbolicequation.solutionarea;

class Subarea {
    final int index;
    final Area parentArea;

    final int xStartIndex;
    final int xEndIndex;

    final int yStartIndex;
    final int yEndIndex;

    Subarea(Area parentArea, int index,
                int xStartIndex, int xEndIndex, int yStartIndex, int yEndIndex) {
        this.parentArea = parentArea;
        this.index = index;

        this.xEndIndex = xEndIndex;
        this.xStartIndex = xStartIndex;
        this.yStartIndex = yStartIndex;
        this.yEndIndex = yEndIndex;
    }

    @Override
    public String toString() {
        return "Subarea: " + index + "; "
                + "start: ("
                + xStartIndex + ", " + yEndIndex + "), end: ("
                + xEndIndex + ", " + yEndIndex + ")" + "; (indexes)";
    }
}

package ru.nstu.hyperbolicequation.solutionarea;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.Scanner;

public class Grid {
    double[] xValues;
    double[] yValues;

    public Grid(Area area, File description) {
        try (Scanner scanner = new Scanner(description)) {
            xValues = getIntervalsBorders(scanner, area.xValues);
            yValues = getIntervalsBorders(scanner, area.yValues);
        } catch (FileNotFoundException e) {
            System.out.println("Error reading file \"" + description.getName() + "\"" + e);
            System.exit(0);
        }
    }
```

```
    public double[] getXValues() {
        return xValues;
    }

    public double[] getYValues() {
        return yValues;
    }

    private double[] getIntervalsBorders(Scanner scanner, double[] sourceValues) {
        int totalBordersQuantity = 0;
        int subareasCount = sourceValues.length - 1;

        int[] intervalsBordersQuantities = new int[subareasCount];
        double[] compressionCoefficients = new double[subareasCount];

        for (int i = 0; i < subareasCount; ++i) {
            int subareaIntervalsQuantity = scanner.nextInt();
            totalBordersQuantity += subareaIntervalsQuantity;

            intervalsBordersQuantities[i] = subareaIntervalsQuantity;
            compressionCoefficients[i] = scanner.nextDouble();
        }

        double[] intervalsBorders = new double[totalBordersQuantity + 1];

        for (int i = 0, index = 0; i < subareasCount; ++i) {
            int intervalBordersQuantity = intervalsBordersQuantities[i];
            double compressionCoefficient = compressionCoefficients[i];

            double startBorder = sourceValues[i];
            double endBorder = sourceValues[i + 1];

            if (compressionCoefficient == 1) {
                double step = (endBorder - startBorder) / intervalBordersQuantity;

                for (int j = 0; j < intervalBordersQuantity; ++j) {
                    intervalsBorders[index + j] = startBorder + step * j;
                }
            } else {
                double base = (endBorder - startBorder)
                        / (Math.pow(compressionCoefficient, intervalBordersQuantity) - 1);

                for (int j = 0; j < intervalBordersQuantity; ++j) {
                    intervalsBorders[index + j] = Math.round((startBorder
                            + base * (Math.pow(compressionCoefficient, j) - 1)) * 100.) /
100.;
                }
            }

            intervalsBorders[index + intervalBordersQuantity] = endBorder;
            index += intervalBordersQuantity;
        }

        return intervalsBorders;
    }

    @Override
    public String toString() {
        return "Grid: "
                + "\t" + System.lineSeparator() + " X: " + Arrays.toString(xValues)
                + "\t" + System.lineSeparator() + " Y: " + Arrays.toString(yValues);
    }
}

package ru.nstu.hyperbolicequation.solutionarea;

import ru.nstu.hyperbolicequation.function.ThreeArityFunction;
import ru.nstu.hyperbolicequation.slae.DenseMatrix;
import ru.nstu.hyperbolicequation.slae.Matrix;
import ru.nstu.hyperbolicequation.slae.Vector;
```

```java
class FiniteElement {
    SolutionArea solutionArea;
    int subAreaIndex;

    double lambdaAverage;
    double gammaAverage;
    double sigmaAverage;
    double chiAverage;

    Vector functionValues = new Vector(NODES_COUNT);

    int xStartIndex;
    int yStartIndex;

    static final int EDGE_NODES_COUNT = 2;
    static final int NODES_COUNT = 4;

    private final Matrix weightMatrixWithGamma1;

    FiniteElement(SolutionArea solutionArea, int xStartIndex, int yStartIndex, Double time) {
        this.solutionArea = solutionArea;

        this.xStartIndex = xStartIndex;
        this.yStartIndex = yStartIndex;

        subAreaIndex = getSubareaIndex();

        if (isSignificant()) {
            lambdaAverage =
getNodesFunctionAverage(solutionArea.area.equation.lambda.get(subAreaIndex), time);
            gammaAverage =
getNodesFunctionAverage(solutionArea.area.equation.gamma.get(subAreaIndex), time);
            sigmaAverage =
getNodesFunctionAverage(solutionArea.area.equation.sigma.get(subAreaIndex), time);
            chiAverage =
getNodesFunctionAverage(solutionArea.area.equation.chi.get(subAreaIndex), time);
            functionValues =
getFunctionValues(solutionArea.area.equation.rightFunctions.get(subAreaIndex), time);
        }

        weightMatrixWithGamma1 = generateWeightMatrix(1);
    }

    FiniteElement(SolutionArea solutionArea, int xStartIndex, int yStartIndex) {
        this(solutionArea, xStartIndex, yStartIndex, 0.);
    }

    public boolean isSignificant() {
        return subAreaIndex >= 0;
    }

    private int getSubareaIndex() {
        double[] gridXValues = solutionArea.grid.xValues;
        double[] gridYValues = solutionArea.grid.yValues;

        double[] areaXValues = solutionArea.area.xValues;
        double[] areaYValues = solutionArea.area.yValues;

        for (int i = 0; i < solutionArea.area.subareas.length; ++i) {
            Subarea subarea = solutionArea.area.subareas[i];

            if (Double.compare(gridXValues[xStartIndex], areaXValues[subarea.xStartIndex]) >=
0
                    && Double.compare(gridXValues[xStartIndex + 1],
areaXValues[subarea.xEndIndex]) <= 0
                    && Double.compare(gridYValues[yStartIndex],
areaYValues[subarea.yStartIndex]) >= 0
                    && Double.compare(gridYValues[yStartIndex + 1],
areaYValues[subarea.yEndIndex]) <= 0) {
                return subarea.index;
```

```
            }
        }

        return -1;
    }

    private double getNodesFunctionAverage(ThreeArityFunction<Double, Double, Double, Double>
function, Double time) {
        double average = 0;

        for (int i = 0; i < EDGE_NODES_COUNT; ++i) {
            for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
                average += function.apply(
                        solutionArea.grid.xValues[xStartIndex + j],
                        solutionArea.grid.yValues[yStartIndex + i],
                        time
                );
            }
        }

        return average / NODES_COUNT;
    }

    private Vector getFunctionValues(ThreeArityFunction<Double, Double, Double, Double>
function, Double time) {
        double[] functionValues = new double[NODES_COUNT];

        for (int i = 0, index = 0; i < EDGE_NODES_COUNT; ++i) {
            for (int j = 0; j < EDGE_NODES_COUNT; ++j, ++index) {
                functionValues[index] = function.apply(
                        solutionArea.grid.xValues[xStartIndex + j],
                        solutionArea.grid.yValues[yStartIndex + i],
                        time
                );
            }
        }

        return new Vector(functionValues);
    }

    Matrix getLocalMatrixCoefficients() {
        Matrix matrix = generateStiffnessMatrix();
        matrix.add(generateWeightMatrix(gammaAverage));
        return matrix;
    }

    Vector getLocalConstantTermsVector() {
        return weightMatrixWithGamma1.multiplyByVector(functionValues);
    }

    public Matrix generateStiffnessMatrix() {
        double heightX = solutionArea.grid.xValues[xStartIndex + 1] -
solutionArea.grid.xValues[xStartIndex];
        double heightY = solutionArea.grid.yValues[yStartIndex + 1] -
solutionArea.grid.yValues[yStartIndex];

        double squaredHeightX = heightX * heightX;
        double squaredHeightY = heightY * heightY;

        final double commonMultiplierConstant = 6;
        double commonMultiplier = lambdaAverage / (commonMultiplierConstant * heightX *
heightY);

        final double[][] leftStiffnessMatrixCoefficients = new double[][]{
                new double[]{2, -2, 1, -1},
                new double[]{-2, 2, -1, 1},
                new double[]{1, -1, 2, -2},
                new double[]{-1, 1, -2, 2}
        };

        final double[][] rightStiffnessMatrixCoefficients = new double[][]{
```

```
                new double[]{2, 1, -2, -1},
                new double[]{1, 2, -1, -2},
                new double[]{-2, -1, 2, 1},
                new double[]{-1, -2, 1, 2}
        };

        final int stiffnessMatrixDimension = leftStiffnessMatrixCoefficients.length;
        Matrix stiffnessMatrix = new DenseMatrix(stiffnessMatrixDimension,
stiffnessMatrixDimension);

        for (int i = 0; i < stiffnessMatrixDimension; ++i) {
            for (int j = 0; j < stiffnessMatrixDimension; ++j) {
                double component = (squaredHeightY * leftStiffnessMatrixCoefficients[i][j]
                        + squaredHeightX * rightStiffnessMatrixCoefficients[i][j])
                        * commonMultiplier;

                stiffnessMatrix.setComponent(i, j, component);
            }
        }

        return stiffnessMatrix;
    }

    public Matrix generateWeightMatrix() {
        return generateWeightMatrix(gammaAverage);
    }

    public Matrix generateWeightMatrix(double gammaMultiplier, double sigmaMultiplier, double
chiMultiplier) {
        double physicalCoefficient = gammaAverage * gammaMultiplier
                + sigmaAverage * sigmaMultiplier
                + chiAverage * chiMultiplier;
        return generateWeightMatrix(physicalCoefficient);
    }

    public Matrix generateWeightMatrix(double physicalCoefficient) {
        final double commonMultiplierConstant = 36;
        double commonMultiplier = physicalCoefficient
                * (solutionArea.grid.xValues[xStartIndex + 1] -
solutionArea.grid.xValues[xStartIndex])
                * (solutionArea.grid.yValues[yStartIndex + 1] -
solutionArea.grid.yValues[yStartIndex])
                / commonMultiplierConstant;

        final double[][] weightMatrixCoefficients = new double[][]{
                new double[]{4, 2, 2, 1},
                new double[]{2, 4, 1, 2},
                new double[]{2, 1, 4, 2},
                new double[]{1, 2, 2, 4}
        };

        final int weightMatrixDimension = weightMatrixCoefficients.length;
        Matrix weightMatrix = new DenseMatrix(weightMatrixDimension, weightMatrixDimension);

        for (int i = 0; i < weightMatrixDimension; ++i) {
            for (int j = 0; j < weightMatrixDimension; ++j) {
                weightMatrix.setComponent(i, j, commonMultiplier *
weightMatrixCoefficients[i][j]);
            }
        }

        return weightMatrix;
    }

    @Override
    public String toString() {
        String separator = System.lineSeparator();
        return String.format("sub: %2d", subAreaIndex) + separator +
                "yStart: " + yStartIndex + separator +
                "xStart: " + xStartIndex;
```

```java
    }
}


package ru.nstu.hyperbolicequation.solutionarea;

class Node {
    int xIndex;
    int yIndex;
    int globalIndex;

    Node(int xIndex, int yIndex, int globalIndex) {
        this.xIndex = xIndex;
        this.yIndex = yIndex;
        this.globalIndex = globalIndex;
    }
}

package ru.nstu.hyperbolicequation.solutionarea;

import ru.nstu.hyperbolicequation.function.ThreeArityFunction;
import ru.nstu.hyperbolicequation.slae.DenseMatrix;
import ru.nstu.hyperbolicequation.slae.InitialTimeFunction;
import ru.nstu.hyperbolicequation.slae.Matrix;
import ru.nstu.hyperbolicequation.slae.Vector;

import java.util.List;

public class HyperbolicEquation {
    List<ThreeArityFunction<Double, Double, Double, Double>> lambda;
    List<ThreeArityFunction<Double, Double, Double, Double>> gamma;
    List<ThreeArityFunction<Double, Double, Double, Double>> sigma;
    List<ThreeArityFunction<Double, Double, Double, Double>> chi;
    List<ThreeArityFunction<Double, Double, Double, Double>> rightFunctions;

    List<ThreeArityFunction<Double, Double, Double, Double>> firstKind;
    List<ThreeArityFunction<Double, Double, Double, Double>> secondKind;
    List<ThreeArityFunction<Double, Double, Double, Double>> thirdKind;

    List<Double> betaCoefficients;

    InitialTimeFunction initialTimeFunction;
    TimeLayers timeLayers;

    private final DescriptionFiles descriptionFiles;

    public HyperbolicEquation(List<ThreeArityFunction<Double, Double, Double, Double>> lambda,
                              List<ThreeArityFunction<Double, Double, Double, Double>> gamma,
                              List<ThreeArityFunction<Double, Double, Double, Double>> sigma,
                              List<ThreeArityFunction<Double, Double, Double, Double>> chi,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
rightFunctions,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
firstKind,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
secondKind,
                              List<ThreeArityFunction<Double, Double, Double, Double>>
thirdKind,
                              List<Double> betaCoefficients,
                              InitialTimeFunction initialTimeFunction,
                              TimeLayers timeLayers,
                              DescriptionFiles descriptionFiles) {
        this.lambda = lambda;
        this.gamma = gamma;
        this.sigma = sigma;
        this.chi = chi;
        this.rightFunctions = rightFunctions;

        this.firstKind = firstKind;
        this.secondKind = secondKind;
        this.thirdKind = thirdKind;
```

```
        this.betaCoefficients = betaCoefficients;

        this.initialTimeFunction = initialTimeFunction;

        this.timeLayers = timeLayers;
        this.descriptionFiles = descriptionFiles;
    }

    public Matrix solve() {
        Area area = new Area(descriptionFiles.getAreaDescription(), this);
        Grid grid = new Grid(area, descriptionFiles.getIntervalsDescription());

        BoundaryConditions conditions = new
BoundaryConditions(descriptionFiles.getBoundariesDescription(),
                firstKind, secondKind, thirdKind, betaCoefficients);

        SolutionArea solutionArea = new SolutionArea(area, grid, conditions, timeLayers,
initialTimeFunction, this);

        int layersCount = timeLayers.countLayers();
        ThreeArityFunction<Double, Double, Double, Double> function =
initialTimeFunction.getFunction();

        int nodesCount = grid.xValues.length * grid.yValues.length;

        Matrix result = solutionArea.solve();
        DenseMatrix matrix = new DenseMatrix(layersCount, nodesCount);

        for (int layer = 0; layer < layersCount; ++layer) {
            double time =  timeLayers.getTime(layer);

            Vector vector = new Vector(nodesCount);

            double nominator = 0;
            double denominator = 0;

            for (int i = 0, index = 0; i < grid.yValues.length; ++i) {
                double y = grid.yValues[i];

                for (int j = 0; j < grid.xValues.length; ++j, ++index) {
                    double x = grid.xValues[j];

                    double functionValue = function.apply(x, y, time);
                    vector.setComponent(index, functionValue);

                   nominator += Math.pow(result.getComponent(layer, index) - functionValue,
2);
                   denominator += functionValue * functionValue;
                }
            }

            nominator = Math.sqrt(nominator);
            denominator = Math.sqrt(denominator);

        return result;
    }
}

package ru.nstu.hyperbolicequation.slae;

import java.util.Arrays;

public class DenseMatrix implements Matrix {
    Vector[] rows;

    public DenseMatrix(int rowsCount, int columnsCount) {
        validateRowsCount(rowsCount);
        validateColumnsCount(columnsCount);

        rows = new Vector[rowsCount];
```

```
        for (int i = 0; i < rowsCount; i++) {
            rows[i] = new Vector(columnsCount);
        }
    }

    public DenseMatrix(DenseMatrix matrixToCopy) {
        int matrixToCopyRowsCount = matrixToCopy.getRowsCount();
        rows = new Vector[matrixToCopyRowsCount];

        for (int i = 0; i < matrixToCopyRowsCount; i++) {
            rows[i] = new Vector(matrixToCopy.rows[i]);
        }
    }

    public DenseMatrix(double[][] array) {
        validateRowsCount(array.length);

        int maxRowLength = 0;

        for (double[] rowArray : array) {
            maxRowLength = Math.max(maxRowLength, rowArray.length);
        }

        validateColumnsCount(maxRowLength);
        rows = new Vector[array.length];

        for (int i = 0; i < array.length; i++) {
            rows[i] = new Vector(maxRowLength, array[i]);
        }
    }

    public DenseMatrix(Vector[] vectorsArray) {
        validateRowsCount(vectorsArray.length);

        int maxRowLength = 0;

        for (Vector vector : vectorsArray) {
            maxRowLength = Math.max(maxRowLength, vector.getSize());
        }

        rows = new Vector[vectorsArray.length];

        for (int i = 0; i < vectorsArray.length; i++) {
            rows[i] = new Vector(maxRowLength);
            rows[i].add(vectorsArray[i]);
        }
    }

    public int getRowsCount() {
        return rows.length;
    }

    public int getColumnsCount() {
        return rows[0].getSize();
    }

    public Vector getRow(int index) {
        validateRowIndex(index);

        return new Vector(rows[index]);
    }

    @Override
    public void resetRow(int index) {
        Arrays.fill(rows[index].components, 0);
    }

    @Override
    public void resetColumn(int index) {
        for (int i = 0; i < rows.length; ++i) {
```

```java
            rows[i].components[index] = 0;
        }
    }

    @Override
    public DenseMatrix assemble() {
        return this;
    }

    public void setRow(int index, Vector row) {
        validateRowIndex(index);

        if (row.getSize() != getColumnsCount()) {
            throw new IllegalArgumentException("The row size must match with the number of the
matrix columns. "
                    + "Matrix columns number: " + getColumnsCount() + ". "
                    + "Current row size: " + row.getSize() + "."
            );
        }

        int rowSize = row.getSize();

        for (int i = 0; i < rowSize; i++) {
            rows[index].setComponent(i, row.getComponent(i));
        }
    }

    public Vector getColumn(int index) {
        if (index < 0 || index >= getColumnsCount()) {
            throw new IndexOutOfBoundsException("The column index out of range. "
                    + "Valid indexes: from 0 to " + (getColumnsCount() - 1) + ". "
                    + "Current index: " + index + "."
            );
        }

        int rowsCount = getRowsCount();
        Vector column = new Vector(rowsCount);

        for (int i = 0; i < rowsCount; i++) {
            column.setComponent(i, rows[i].getComponent(index));
        }

        return column;
    }

    public double getComponent(int rowIndex, int columnIndex) {
        return rows[rowIndex].getComponent(columnIndex);
    }

    @Override
    public void setComponent(int rowIndex, int columnIndex, double component) {
        rows[rowIndex].setComponent(columnIndex, component);
    }

    @Override
    public void increaseComponent(int rowIndex, int columnIndex, double addition) {
        double component = getComponent(rowIndex, columnIndex) + addition;
        rows[rowIndex].setComponent(columnIndex, component);
    }

    public void transpose() {
        int rowsCount = getRowsCount();
        int columnsCount = getColumnsCount();

        if (rowsCount == columnsCount) {
            for (int i = 0; i < rowsCount; i++) {
                for (int j = 0; j < i; j++) {
                    double temp = rows[i].getComponent(j);
                    rows[i].setComponent(j, rows[j].getComponent(i));
                    rows[j].setComponent(i, temp);
                }
```

```
            }
        } else {
            Vector[] transposedRows = new Vector[columnsCount];

            for (int i = 0; i < columnsCount; ++i) {
                transposedRows[i] = getColumn(i);
            }

            rows = transposedRows;
        }
    }

    public double getDeterminant() {
        int rowsCount = getRowsCount();
        int columnsCount = getColumnsCount();

        if (rowsCount != columnsCount) {
            throw new UnsupportedOperationException("Matrix must be squared. "
                    + "Current rows count: " + rowsCount + ". "
                    + " Current columns count: " + columnsCount + "."
            );
        }

        return getDeterminant(this);
    }

    private static double getDeterminant(DenseMatrix matrix) {
        int size = matrix.getRowsCount();

        if (size == 1) {
            return matrix.rows[0].getComponent(0);
        }

        if (size == 2) {
            return matrix.rows[0].getComponent(0) * matrix.rows[1].getComponent(1)
                    - matrix.rows[0].getComponent(1) * matrix.rows[1].getComponent(0);
        }

        double determinant = 0;

        for (int i = 0; i < size; ++i) {
            determinant += matrix.rows[0].getComponent(i)
                    * Math.pow(-1, i)
                    * getMinorMatrix(matrix, i).getDeterminant();
        }

        return determinant;
    }

    private static DenseMatrix getMinorMatrix(DenseMatrix matrix, int removedColumnIndex) {
        int oldSize = matrix.getRowsCount();
        int newSize = oldSize - 1;

        DenseMatrix minorMatrix = new DenseMatrix(newSize, newSize);

        for (int i = 1; i < oldSize; ++i) {
            Vector row = matrix.rows[i];

            for (int j = 0, k = 0; j < oldSize; ++j) {
                if (j != removedColumnIndex) {
                    minorMatrix.rows[i - 1].setComponent(k, row.getComponent(j));
                    ++k;
                }
            }
        }

        return minorMatrix;
    }

    public void multiplyByScalar(double scalar) {
        for (Vector row : rows) {
```

```
            row.multiplyByScalar(scalar);
        }
    }

    public void add(Matrix matrix) {
        validateSizesEquality(matrix);

        int rowsCount = getRowsCount();

        for (int i = 0; i < rowsCount; i++) {
            rows[i].add(matrix.getRow(i));
        }
    }

    public void subtract(DenseMatrix matrix) {
        validateSizesEquality(matrix);

        int rowsCount = getRowsCount();

        for (int i = 0; i < rowsCount; i++) {
            rows[i].subtract(matrix.rows[i]);
        }
    }

    public Vector multiplyByVector(Vector vector) {
        if (getColumnsCount() != vector.getSize()) {
            throw new IllegalArgumentException("The size of the column vector must be equal to
the number of matrix columns. "
                    + "Matrix columns count: " + getColumnsCount() + ". "
                    + "Column vector size: " + vector.getSize() + "."
            );
        }

        int rowsCount = getRowsCount();
        Vector resultVector = new Vector(rowsCount);

        for (int i = 0; i < rowsCount; i++) {
            resultVector.setComponent(i, Vector.getDotProduct(rows[i], vector));
        }

        return resultVector;
    }

    private static void validateRowsCount(int rowsCount) {
        if (rowsCount <= 0) {
            throw new IllegalArgumentException("Matrix rows count must be positive. "
                    + "Current rows count: " + rowsCount + "."
            );
        }
    }

    private static void validateColumnsCount(int columnsCount) {
        if (columnsCount <= 0) {
            throw new IllegalArgumentException("Matrix columns count must be positive. "
                    + "Current columns count: " + columnsCount + "."
            );
        }
    }

    private void validateRowIndex(int index) {
        if (index < 0 || index >= rows.length) {
            throw new IndexOutOfBoundsException("The row index out of range. "
                    + "Valid indexes: from 0 to " + (rows.length - 1) + "."
                    + "Current index: " + index + "."
            );
        }
    }

    private void validateSizesEquality(Matrix matrix) {
        if (getRowsCount() != matrix.getRowsCount() || getColumnsCount() !=
matrix.getColumnsCount()) {
```

```
            throw new IllegalArgumentException("For arithmetic operations, the matrices must
be the same size. "
                    + "Current matrix rows count: " + getRowsCount() + ", "
                    + "columns count: " + getColumnsCount() + ". "
                    + "Passed matrix rows count: " + matrix.getRowsCount() + ", "
                    + "columns count: " + matrix.getColumnsCount() + "."
            );
        }
    }

    public static DenseMatrix getSum(DenseMatrix matrix1, DenseMatrix matrix2) {
        matrix1.validateSizesEquality(matrix2);

        DenseMatrix resultMatrix = new DenseMatrix(matrix1);
        resultMatrix.add(matrix2);

        return resultMatrix;
    }

    public static DenseMatrix getDifference(DenseMatrix matrix1, DenseMatrix matrix2) {
        matrix1.validateSizesEquality(matrix2);

        DenseMatrix resultMatrix = new DenseMatrix(matrix1);
        resultMatrix.subtract(matrix2);

        return resultMatrix;
    }

    public static DenseMatrix getProduct(DenseMatrix matrix1, DenseMatrix matrix2) {
        if (matrix1.getColumnsCount() != matrix2.getRowsCount()) {
            throw new IllegalArgumentException("For the product columns the number of the 1-st
matrix "
                    + "must be equal to the rows number of the 2-nd matrix. "
                    + "First matrix columns count: " + matrix1.getColumnsCount() + ". "
                    + "Second matrix rows count: " + matrix2.getRowsCount() + "."
            );
        }

        int rowsCount = matrix1.getRowsCount();
        int columnsCount = matrix2.getColumnsCount();

        DenseMatrix resultMatrix = new DenseMatrix(rowsCount, columnsCount);

        for (int i = 0; i < rowsCount; i++) {
            Vector row = matrix1.rows[i];

            for (int j = 0; j < columnsCount; j++) {
                Vector column = matrix2.getColumn(j);

                resultMatrix.rows[i].setComponent(j, Vector.getDotProduct(row, column));
            }
        }

        return resultMatrix;
    }

    public static DenseMatrix getProduct(DenseMatrix matrix, double scalar) {
        DenseMatrix resultMatrix = new DenseMatrix(matrix);
        resultMatrix.multiplyByScalar(scalar);

        return resultMatrix;
    }

    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append('{').append(System.lineSeparator());

        final String separator = " " + System.lineSeparator();

        for (Vector row : rows) {
```

```
            stringBuilder.append(row).append(separator);
        }

        stringBuilder.delete(stringBuilder.length() - separator.length(),
stringBuilder.length());
        return stringBuilder.append(System.lineSeparator()).append('}').toString();
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }

        if (o == null || o.getClass() != getClass()) {
            return false;
        }

        DenseMatrix matrix = (DenseMatrix) o;

        if (rows.length != matrix.rows.length || getColumnsCount() !=
matrix.getColumnsCount()) {
            return false;
        }

        for (int i = 0; i < rows.length; i++) {
            if (!rows[i].equals(matrix.rows[i])) {
                return false;
            }
        }

        return true;
    }

    @Override
    public int hashCode() {
        final int prime = 17;
        int hash = 1;

        for (Vector row : rows) {
            hash = prime * hash + row.hashCode();
        }

        return hash;
    }
}

package ru.nstu.hyperbolicequation.slae;

import ru.nstu.hyperbolicequation.function.ThreeArityFunction;

public class InitialTimeFunction {
    private final ThreeArityFunction<Double, Double, Double, Double> function;
    private final ThreeArityFunction<Double, Double, Double, Double> partialDerivative;

    public InitialTimeFunction(ThreeArityFunction<Double, Double, Double, Double> function,
                               ThreeArityFunction<Double, Double, Double, Double>
partialDerivative) {
        this.function = function;
        this.partialDerivative = partialDerivative;
    }

    public ThreeArityFunction<Double, Double, Double, Double> getFunction() {
        return function;
    }

    public ThreeArityFunction<Double, Double, Double, Double> getPartialDerivative() {
        return partialDerivative;
    }

    public double calculate(double x, double y, double t0) {
```

```java
        return function.apply(x, y, t0);
    }

    public double calculatePartialDerivative(double x, double y, double t0) {
        return partialDerivative.apply(x, y, t0);
    }

    public double calculateFirstTimeFunction(double x, double y, double t0, double t1) {
        return calculate(x, y, t1);

    }
}
package ru.nstu.hyperbolicequation.slae;

public interface Matrix {
    int getRowsCount();

    int getColumnsCount();

    double getComponent(int rowIndex, int columnIndex);

    void setComponent(int rowIndex, int columnIndex, double component);

    Vector getRow(int index);

    void resetRow(int index);

    void resetColumn(int index);

    void add(Matrix matrix);

    DenseMatrix assemble();

    void increaseComponent(int rowIndex, int columnIndex, double component);

    Vector multiplyByVector(Vector vector);
}


package ru.nstu.hyperbolicequation.slae;

import java.util.Arrays;

public class SparseRowMatrix implements Matrix {
    int rowsCount;
    int columnsCount;

    double[] values;
    int[] columnsIndexes;
    int[] pointers;

    public SparseRowMatrix(double[] values, int[] columnsIndexes, int[] pointers) {
        this.values = values;
        this.columnsIndexes = columnsIndexes;
        this.pointers = pointers;

        rowsCount = pointers.length - 1;
        int maxIndex = -1;

        for (int index : columnsIndexes) {
            if (index > maxIndex) {
                maxIndex = index;
            }
        }

        columnsCount = maxIndex + 1;
    }

    public SparseRowMatrix(double[][] components) {
        rowsCount = components.length;
        columnsCount = components[0].length;
```

```
        int maxNodesCount = rowsCount * columnsCount;

        values = new double[maxNodesCount];
        columnsIndexes = new int[maxNodesCount];
        pointers = new int[rowsCount + 1];

        for (int i = 0, index = 0; i < rowsCount; ++i) {
            pointers[i + 1] = pointers[i];

            for (int j = 0; j < columnsCount; ++j) {
                double component = components[i][j];

                if (Double.compare(component, 0) != 0) {
                    values[index] = component;
                    columnsIndexes[index] = j;
                    ++pointers[i + 1];
                    ++index;
                }
            }
        }

        values = Arrays.copyOf(values, pointers[rowsCount]);
        columnsIndexes = Arrays.copyOf(columnsIndexes, pointers[rowsCount]);
    }

    public DenseMatrix assemble() {
        double[][] components = new double[rowsCount][columnsCount];

        for (int i = 0, pointer = 0; i < rowsCount; ++i) {
            int elementsCount = pointers[i + 1] - pointers[i];

            for (int j = 0; j < elementsCount; ++j) {
                components[i][columnsIndexes[pointer]] = values[pointer];
                ++pointer;
            }
        }

        return new DenseMatrix(components);
    }

    @Override
    public String toString() {
        String separator = System.lineSeparator();
        return "Sparse row matrix: " + rowsCount + 'x' + columnsCount + separator +
                "pointers:" + System.lineSeparator() + Arrays.toString(pointers) + separator +
                "values: " + System.lineSeparator() + Arrays.toString(values) + separator +
                "columns indexes:" + System.lineSeparator() + Arrays.toString(columnsIndexes)
+ separator;
    }

    @Override
    public int getRowsCount() {
        return rowsCount;
    }

    @Override
    public int getColumnsCount() {
        return columnsCount;
    }

    @Override
    public double getComponent(int rowIndex, int columnIndex) {
        int elementsCount = pointers[rowIndex + 1] - pointers[rowIndex];

        for (int i = 0; i < elementsCount; ++i) {
            if (columnsIndexes[rowIndex + i] == columnIndex) {
                return values[rowIndex + i];
            }
        }

        throw new IndexOutOfBoundsException();
```

```
}

@Override
public void setComponent(int rowIndex, int columnIndex, double component) {
    int elementsCount = pointers[rowIndex + 1] - pointers[rowIndex];

    for (int i = 0; i < elementsCount; ++i) {
        if (columnsIndexes[pointers[rowIndex] + i] == columnIndex) {
            values[pointers[rowIndex] + i] = component;
            return;
        }
    }

    throw new IndexOutOfBoundsException();
}

@Override
public void resetRow(int index) {
    int elementsCount = pointers[index + 1] - pointers[index];

    for (int i = 0; i < elementsCount; ++i) {
        values[pointers[index] + i] = 0;
    }
}

@Override
public void resetColumn(int index) {
    for (int i = 0, j = 0; i < columnsIndexes.length || j == columnsCount; ++i) {
        if (columnsIndexes[i] == index) {
            values[i] = 0;
            ++j;
        }
    }
}

@Override
public void increaseComponent(int rowIndex, int columnIndex, double component) {
    int elementsCount = pointers[rowIndex + 1] - pointers[rowIndex];

    for (int i = 0; i < elementsCount; ++i) {
        if (columnsIndexes[pointers[rowIndex] + i] == columnIndex) {
            values[pointers[rowIndex] + i] += component;
            return;
        }
    }
}

private void add(int rowIndex, int columnIndex, double component) {
    int[] newColumnsIndexes = new int[columnsIndexes.length + 1];
    double[] newValues = new double[values.length + 1];

    System.arraycopy(columnsIndexes, 0, newColumnsIndexes, 0, pointers[rowIndex]);
    System.arraycopy(columnsIndexes, pointers[rowIndex], newColumnsIndexes,
            pointers[rowIndex] + 1, pointers[rowsCount] - pointers[rowIndex]);
    newColumnsIndexes[pointers[rowIndex]] = columnIndex;

    System.arraycopy(values, 0, newValues, 0, pointers[rowIndex]);
    System.arraycopy(values, pointers[rowIndex], newValues,
            pointers[rowIndex] + 1, pointers[rowsCount] - pointers[rowIndex]);
    newValues[pointers[rowIndex]] = component;

    for (int i = rowIndex + 1; i < pointers.length; ++i) {
        ++pointers[i];
    }

    values = newValues;
    columnsIndexes = newColumnsIndexes;
}

@Override
public Vector multiplyByVector(Vector vector) {
```

```
        double[] components = new double[rowsCount];

        for (int i = 1; i < pointers.length; ++i) {
            int rowIndex = i - 1;
            int elementsCount = pointers[i] - pointers[rowIndex];
            int startIndex = pointers[rowIndex];

            for (int j = 0; j < elementsCount; ++j) {
                components[rowIndex] += values[startIndex + j] *
vector.components[columnsIndexes[startIndex + j]];
            }
        }

        return new Vector(components);
    }

    @Override
    public void add(Matrix matrix) {
    }

    @Override
    public Vector getRow(int index) {
        return null;
    }
}

package ru.nstu.hyperbolicequation.slae;

import java.util.Arrays;

public class Vector {
    double[] components;

    public Vector(int size) {
        validateSize(size);

        components = new double[size];
    }

    public Vector(Vector vectorToCopy) {
        components = Arrays.copyOf(vectorToCopy.components, vectorToCopy.components.length);
    }

    public Vector(double[] components) {
        validateSize(components.length);

        this.components = Arrays.copyOf(components, components.length);
    }

    public Vector(int size, double[] components) {
        validateSize(size);

        this.components = Arrays.copyOf(components, size);
    }

    public int getSize() {
        return components.length;
    }

    public void add(Vector vector) {
        if (components.length < vector.components.length) {
            components = Arrays.copyOf(components, vector.components.length);
        }

        for (int i = 0; i < vector.components.length; i++) {
            components[i] += vector.components[i];
        }
    }

    public void subtract(Vector vector) {
        if (components.length < vector.components.length) {
```

```
            components = Arrays.copyOf(components, vector.components.length);
        }

        for (int i = 0; i < vector.components.length; i++) {
            components[i] -= vector.components[i];
        }
    }

    public void multiplyByScalar(double scalar) {
        for (int i = 0; i < components.length; i++) {
            components[i] *= scalar;
        }
    }

    public double getComponent(int index) {
        validateIndex(index);

        return components[index];
    }

    public void setComponent(int index, double component) {
        validateIndex(index);

        components[index] = component;
    }

    public void increaseComponent(int index, double addition) {
        validateIndex(index);
        components[index] += addition;
    }

    public static double getDotProduct(Vector vector1, Vector vector2) {
        int minSize = Math.min(vector1.components.length, vector2.components.length);

        double dotProduct = 0;

        for (int i = 0; i < minSize; i++) {
            dotProduct += vector1.components[i] * vector2.components[i];
        }

        return dotProduct;
    }

    private void validateSize(int size) {
        if (size <= 0) {
            throw new IllegalArgumentException("The size of the vector must be positive. "
                    + "Current size: " + size);
        }
    }

    private void validateIndex(int index) {
        if (index < 0 || index >= components.length) {
            throw new IndexOutOfBoundsException("Index out of range. "
                    + "Valid index: from 0 to " + (components.length - 1) + ". "
                    + "Current index: " + index);
        }
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }

        if (o == null || o.getClass() != getClass()) {
            return false;
        }

        Vector vector = (Vector) o;

        return Arrays.equals(components, vector.components);
```

```java
    }

    @Override
    public int hashCode() {
        final int prime = 37;

        return prime + Arrays.hashCode(components);
    }


    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(' ');

        for (double component : components) {
            stringBuilder.append(String.format("%20.16f", component)).append(" ");
        }

        final int componentsSeparatorLength = 2;
        stringBuilder.delete(stringBuilder.length() - componentsSeparatorLength,
stringBuilder.length());

        return stringBuilder.append(' ').toString();
    }
}


package ru.nstu.hyperbolicequation.solutionarea;

import ru.nstu.hyperbolicequation.function.ThreeArityFunction;
import ru.nstu.hyperbolicequation.slae.*;

import java.util.Arrays;
import java.util.List;

import static ru.nstu.hyperbolicequation.solutionarea.FiniteElement.EDGE_NODES_COUNT;
import static ru.nstu.hyperbolicequation.solutionarea.FiniteElement.NODES_COUNT;

public class SolutionArea {
    Area area;
    Grid grid;

    HyperbolicEquation equation;
    BoundaryConditions conditions;

    TimeLayers timeLayers;
    InitialTimeFunction initialTimeFunction;

    DenseMatrix GlobalStiffnessMatrix;

    public SolutionArea(Area area,
                        Grid grid,
                        BoundaryConditions conditions,
                        TimeLayers timeLayers,
                        InitialTimeFunction initialTimeFunction,
                        HyperbolicEquation equation) {
        this.area = area;
        this.grid = grid;
        this.conditions = conditions;
        this.timeLayers = timeLayers;
        this.initialTimeFunction = initialTimeFunction;
        this.equation = equation;
    }

public Matrix solve() {
    int nodesCount = grid.xValues.length * grid.yValues.length;

    Vector result1 = new Vector(nodesCount);
    Vector result2 = new Vector(nodesCount);
```

```
    double layer0Time = timeLayers.getTime(0);
    double layer1Time = timeLayers.getTime(1);

    for (int i = 0, index = 0; i < grid.yValues.length; ++i) {
        double y = grid.yValues[i];

        for (int j = 0; j < grid.xValues.length; ++j, ++index) {
            double x = grid.xValues[j];

            result2.setComponent(index, initialTimeFunction.calculate(x, y, layer0Time));
            result1.setComponent(index, initialTimeFunction.calculateFirstTimeFunction(x, y,
layer0Time, layer1Time));
        }
    }

    DenseMatrix stiffnessMatrix = getGlobalStiffnessMatrix();
    setThirdConditionsOnMatrix(stiffnessMatrix);
    DenseMatrix halfStiffnessMatrix = DenseMatrix.getProduct(stiffnessMatrix, 0.5);

    DenseMatrix resultMatrix = new DenseMatrix(timeLayers.countLayers(), nodesCount);

    resultMatrix.setRow(0, new Vector(result2));
    resultMatrix.setRow(1, new Vector(result1));

    Vector constantTerms2 = getGlobalConstantTermsVector(layer0Time);

    setSecondConditions(constantTerms2, layer0Time);
    setThirdConditionsOnVector(constantTerms2, layer0Time);

    Vector constantTerms1 = getGlobalConstantTermsVector(layer1Time);

    setSecondConditions(constantTerms1, layer1Time);
    setThirdConditionsOnVector(constantTerms1, layer1Time);

    int layersCount = timeLayers.countLayers();

    for (int layer = 2; layer < layersCount; ++layer) {
        double time0 = timeLayers.getTime(layer);
        double time1 = timeLayers.getTime(layer - 1);
        double time2 = timeLayers.getTime(layer - 2);

        double delta1 = time0 - time2;
        double delta2 = (time0 * (time0 - time1 - time2) + time1 * time2) / 2.;

        double ratio1 = (time0 - time2) / (time1 - time2);
        double ratio2 = (time0 - time1) / (time1 - time2);

        DenseMatrix weightMatrixWithCoefficients = getGlobalWeightMatrix(0.5, 1 / delta1, 1 /
delta2);
        DenseMatrix systemCoefficients = DenseMatrix.getSum(halfStiffnessMatrix,
weightMatrixWithCoefficients);

        Vector constantTerms0 = getGlobalConstantTermsVector(time0);

        setSecondConditions(constantTerms0, time0);
        setThirdConditionsOnVector(constantTerms0, time0);

        Vector constantTerms = new Vector(constantTerms0);

        constantTerms.add(constantTerms2);
        constantTerms.multiplyByScalar(0.5);

        Vector tempVector = new DenseMatrix(halfStiffnessMatrix).multiplyByVector(result2);

        constantTerms.subtract(tempVector);

        Matrix tempMatrix = getGlobalWeightMatrix(0, 0, ratio1 / delta2);
        constantTerms.add(tempMatrix.multiplyByVector(result1));

        tempMatrix = getGlobalWeightMatrix(-0.5, 1 / delta1, -ratio2 / delta2);
        constantTerms.add(tempMatrix.multiplyByVector(result2));
```

```java
        constantTerms2 = new Vector(constantTerms1);
        constantTerms1 = new Vector(constantTerms0);

        Vector result = new Slae(systemCoefficients, constantTerms).solve();

        result2 = new Vector(result1);
        result1 = new Vector(result);

        resultMatrix.setRow(layer, new Vector(result));

        System.out.println("Time: " + time0);
    }

    return resultMatrix;
}

private Matrix getGlobalStiffnessMatrix() {
    if (GlobalStiffnessMatrix != null) {
        return GlobalStiffnessMatrix;
    }

    int nodesCount = grid.xValues.length * grid.yValues.length;

    double[][] matrixComponents = new double[nodesCount][nodesCount];

    int yValuesLastIndex = grid.yValues.length - 1;
    int xValuesLastIndex = grid.xValues.length - 1;

    for (int i = 0; i < yValuesLastIndex; ++i) {
        for (int j = 0; j < xValuesLastIndex; ++j) {
            FiniteElement element = new FiniteElement(this, j, i);
            Node[] nodes = new Node[NODES_COUNT];

            Matrix localStiffnessMatrix = element.generateStiffnessMatrix();

            for (int k = 0, index = 0; k < EDGE_NODES_COUNT; ++k) {
                int yIndex = element.yStartIndex + k;

                for (int l = 0; l < EDGE_NODES_COUNT; ++index, ++l) {
                    int xIndex = element.xStartIndex + l;

                    nodes[index] = new Node(xIndex, yIndex, getNodeGlobalIndex(xIndex,
yIndex));
                }
            }

            for (int k = 0; k < NODES_COUNT; ++k) {
                for (int l = 0; l < NODES_COUNT; ++l) {
                    matrixComponents[nodes[k].globalIndex][nodes[l].globalIndex] +=
localStiffnessMatrix.getComponent(k, l);
                }
            }
        }
    }

    GlobalStiffnessMatrix = new SparseRowMatrix(matrixComponents);
    return GlobalStiffnessMatrix;
}

private Matrix getGlobalWeightMatrix(double gammaMultiplier, double sigmaMultiplier, double
chiMultiplier) {
    int nodesCount = grid.xValues.length * grid.yValues.length;

    double[][] matrixComponents = new double[nodesCount][nodesCount];

    int yValuesLastIndex = grid.yValues.length - 1;
    int xValuesLastIndex = grid.xValues.length - 1;

    for (int i = 0; i < yValuesLastIndex; ++i) {
        for (int j = 0; j < xValuesLastIndex; ++j) {
```

```
            FiniteElement element = new FiniteElement(this, j, i);
            Node[] nodes = new Node[NODES_COUNT];

            Matrix localWeightMatrix = element.generateWeightMatrix(gammaMultiplier,
sigmaMultiplier, chiMultiplier);

            for (int k = 0, index = 0; k < EDGE_NODES_COUNT; ++k) {
                int yIndex = element.yStartIndex + k;

                for (int l = 0; l < EDGE_NODES_COUNT; ++index, ++l) {
                    int xIndex = element.xStartIndex + l;

                    nodes[index] = new Node(xIndex, yIndex, getNodeGlobalIndex(xIndex,
yIndex));
                }
            }

            for (int k = 0; k < NODES_COUNT; ++k) {
                for (int l = 0; l < NODES_COUNT; ++l) {
                    matrixComponents[nodes[k].globalIndex][nodes[l].globalIndex] +=
localWeightMatrix.getComponent(k, l);
                }
            }
        }
    }

    return new SparseRowMatrix(matrixComponents);
}


private int getNodeGlobalIndex(int xIndex, int yIndex) {
    return yIndex * grid.xValues.length + xIndex;
}

private void setConditions(Matrix matrix, Vector vector, double time) {
    setSecondConditions(vector, time);

    setThirdConditionsOnMatrix(matrix);
    setThirdConditionsOnVector(vector, time);

    setFirstConditionsOnMatrix(matrix);
    setFirstConditionsOnVector(vector, time);
}


private void setFirstConditionsOnMatrix(Matrix matrix) {
    List<List<Integer>> edgesType1 =
conditions.edges.get(BoundaryConditions.FIRST_CONDITION_NUMBER);

    for (List<Integer> edge : edgesType1) {
        int edgeIndex = edge.getFirst();
        ThreeArityFunction<Double, Double, Double, Double> function =
conditions.firstKind.get(edgeIndex);

        int xStart = edge.get(BoundaryConditions.X_START_POSITION);
        int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

        int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
        int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

        if (xEnd == xStart) {
            for (int i = yStart + 1; i <= yEnd; ++i) {
                setFirstConditionsOnMatrix(xStart, xEnd, i - 1, i, matrix);
            }
        } else {
            for (int i = xStart + 1; i <= xEnd; ++i) {
                setFirstConditionsOnMatrix(i - 1, i, yStart, yEnd, matrix);
            }
        }
    }
}
```

```java
private void setFirstConditionsOnMatrix(int xStart, int xEnd, int yStart, int yEnd, Matrix
matrix) {
    int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

    nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
    nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

    matrix.resetRow(nodesGlobalIndexes[0]);
    matrix.resetRow(nodesGlobalIndexes[1]);

    matrix.setComponent(nodesGlobalIndexes[0], nodesGlobalIndexes[0], 1);
    matrix.setComponent(nodesGlobalIndexes[1], nodesGlobalIndexes[1], 1);
}

private void setFirstConditionsOnVector(Vector vector, double time) {
    List<List<Integer>> edgesType1 =
conditions.edges.get(BoundaryConditions.FIRST_CONDITION_NUMBER);

    for (List<Integer> edge : edgesType1) {
        int edgeIndex = edge.getFirst();
        ThreeArityFunction<Double, Double, Double, Double> function =
conditions.firstKind.get(edgeIndex);

        int xStart = edge.get(BoundaryConditions.X_START_POSITION);
        int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

        int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
        int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

        if (xEnd == xStart) {
            for (int i = yStart + 1; i <= yEnd; ++i) {
                setFirstConditionsOnVector(xStart, xEnd, i - 1, i, vector, function, time);
            }
        } else {
            for (int i = xStart + 1; i <= xEnd; ++i) {
                setFirstConditionsOnVector(i - 1, i, yStart, yEnd, vector, function, time);
            }
        }
    }
}

private void setFirstConditionsOnVector(int xStart, int xEnd, int yStart, int yEnd, Vector
vector,
                                        ThreeArityFunction<Double, Double, Double, Double>
function, double time) {
    int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

    nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
    nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

    vector.setComponent(nodesGlobalIndexes[0], function.apply(grid.xValues[xStart],
grid.yValues[yStart], time));
    vector.setComponent(nodesGlobalIndexes[1], function.apply(grid.xValues[xEnd],
grid.yValues[yEnd], time));
}

private void setFirstConditions(Matrix matrix, Vector vector, double time) {
    List<List<Integer>> edgesType1 =
conditions.edges.get(BoundaryConditions.FIRST_CONDITION_NUMBER);

    for (List<Integer> edge : edgesType1) {
        int edgeIndex = edge.getFirst();
        ThreeArityFunction<Double, Double, Double, Double> function =
conditions.firstKind.get(edgeIndex);

        int xStart = edge.get(BoundaryConditions.X_START_POSITION);
        int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

        int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
        int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);
```

```
        if (xEnd == xStart) {
            for (int i = yStart + 1; i <= yEnd; ++i) {
                setFirstConditionOnElement(xStart, xEnd, i - 1, i, matrix, vector, function,
time);
            }
        } else {
            for (int i = xStart + 1; i <= xEnd; ++i) {
                setFirstConditionOnElement(i - 1, i, yStart, yEnd, matrix, vector, function,
time);
            }
        }
    }
}

private void setSecondConditions(Vector vector, double time) {
    List<List<Integer>> edgesType2 =
conditions.edges.get(BoundaryConditions.SECOND_CONDITION_NUMBER);

    for (List<Integer> edge : edgesType2) {
        ThreeArityFunction<Double, Double, Double, Double> thetaFunction =
conditions.secondKind.get(edge.getFirst());

        int xStart = edge.get(BoundaryConditions.X_START_POSITION);
        int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

        int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
        int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

        if (xEnd == xStart) {
            for (int i = yStart + 1; i <= yEnd; ++i) {
                int yElementStart = i - 1;
                setSecondConditionOnElement(xStart, xEnd, yElementStart, i,
                        grid.yValues[i] - grid.yValues[yElementStart], vector, thetaFunction,
time);
            }
        } else {
            for (int i = xStart + 1; i <= xEnd; ++i) {
                int xElementStart = i - 1;
                setSecondConditionOnElement(xElementStart, i, yStart, yEnd,
                        grid.xValues[i] - grid.xValues[xElementStart], vector, thetaFunction,
time);
            }
        }
    }
}

private void setThirdConditionsOnMatrix(Matrix matrix) {
    List<List<Integer>> edgesType3 =
conditions.edges.get(BoundaryConditions.THIRD_CONDITION_NUMBER);

    for (List<Integer> edge : edgesType3) {
        int edgeIndex = edge.getFirst();

        double beta = conditions.beta.get(edgeIndex);

        int xStart = edge.get(BoundaryConditions.X_START_POSITION);
        int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

        int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
        int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

        if (xEnd == xStart) {
            for (int i = yStart + 1; i <= yEnd; ++i) {
                int yElementStart = i - 1;
                double commonMultiplier = beta * (grid.yValues[i] -
grid.yValues[yElementStart])
                        / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;
                setThirdConditionsOnMatrix(xStart, xEnd, yElementStart, i, commonMultiplier,
matrix);
            }
```

```
        } else {
            for (int i = xStart + 1; i <= xEnd; ++i) {
                int xElementStart = i - 1;
                double commonMultiplier = beta * (grid.xValues[i] -
grid.xValues[xElementStart])
                        / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

                setThirdConditionsOnMatrix(xElementStart, i, yStart, yEnd, commonMultiplier,
matrix);
            }
        }
    }
}

private void setThirdConditionsOnMatrix(int xStart, int xEnd, int yStart, int yEnd,
                                        double commonMultiplier, Matrix matrix) {
    int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

    nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
    nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

    DenseMatrix localMatrix = new DenseMatrix(BoundaryConditions.LOCAL_MATRIX);
    localMatrix.multiplyByScalar(commonMultiplier);

    for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
        for (int k = 0; k < EDGE_NODES_COUNT; ++k) {
            matrix.increaseComponent(nodesGlobalIndexes[j], nodesGlobalIndexes[k],
localMatrix.getComponent(j, k));
        }
    }
}

private void setThirdConditionsOnVector(Vector vector, double time) {
    List<List<Integer>> edgesType3 =
conditions.edges.get(BoundaryConditions.THIRD_CONDITION_NUMBER);

    for (List<Integer> edge : edgesType3) {
        int edgeIndex = edge.getFirst();

        ThreeArityFunction<Double, Double, Double, Double> betaFunction =
conditions.thirdKind.get(edgeIndex);
        double beta = conditions.beta.get(edgeIndex);

        int xStart = edge.get(BoundaryConditions.X_START_POSITION);
        int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

        int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
        int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

        if (xEnd == xStart) {
            for (int i = yStart + 1; i <= yEnd; ++i) {
                int yElementStart = i - 1;
                double commonMultiplier = beta * (grid.yValues[i] -
grid.yValues[yElementStart])
                        / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

                setThirdConditionsOnVector(xStart, xEnd, yElementStart, i, commonMultiplier,
                        vector, betaFunction, time);
            }
        } else {
            for (int i = xStart + 1; i <= xEnd; ++i) {
                int xElementStart = i - 1;
                double commonMultiplier = beta * (grid.xValues[i] -
grid.xValues[xElementStart])
                        / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

                setThirdConditionsOnVector(xElementStart, i, yStart, yEnd, commonMultiplier,
                        vector, betaFunction, time);
            }
        }
    }
```

```
}

private void setThirdConditionsOnVector(int xStart, int xEnd, int yStart, int yEnd,
                                        double commonMultiplier, Vector vector,
                                        ThreeArityFunction<Double, Double, Double, Double>
function, double time) {
    double[] vectorComponents = new double[EDGE_NODES_COUNT];
    int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

    nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
    nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

    vectorComponents[0] = commonMultiplier
            * function.apply(grid.xValues[xStart], grid.yValues[yStart], time);
    vectorComponents[1] = commonMultiplier
            * function.apply(grid.xValues[xEnd], grid.yValues[yEnd], time);

    Vector localVector = BoundaryConditions.LOCAL_MATRIX.multiplyByVector(new
Vector(vectorComponents));

    for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
        vector.increaseComponent(nodesGlobalIndexes[j], localVector.getComponent(j));
    }
}

    private void setFirstConditionOnElement(int xStart, int xEnd, int yStart, int yEnd,
                                            Matrix matrix, Vector vector,
                                            ThreeArityFunction<Double, Double, Double, Double>
function, double time) {
        int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

        nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
        nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

        matrix.resetRow(nodesGlobalIndexes[0]);
        matrix.resetRow(nodesGlobalIndexes[1]);

        matrix.setComponent(nodesGlobalIndexes[0], nodesGlobalIndexes[0], 1);
        matrix.setComponent(nodesGlobalIndexes[1], nodesGlobalIndexes[1], 1);

        vector.setComponent(nodesGlobalIndexes[0], function.apply(grid.xValues[xStart],
grid.yValues[yStart], time));
        vector.setComponent(nodesGlobalIndexes[1], function.apply(grid.xValues[xEnd],
grid.yValues[yEnd], time));
    }

    private void setSecondConditionOnElement(int xStart, int xEnd, int yStart, int yEnd,
                                             double elementEdgeLength, Vector vector,
                                             ThreeArityFunction<Double, Double, Double,
Double> function, double time) {
        double[] theta = new double[EDGE_NODES_COUNT];
        int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

        nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
        nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

        theta[0] = elementEdgeLength
                * function.apply(grid.xValues[xStart], grid.yValues[yStart], time)
                / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;
        theta[1] = elementEdgeLength
                * function.apply(grid.xValues[xEnd], grid.yValues[yEnd], time)
                / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

        Vector localVector = BoundaryConditions.LOCAL_MATRIX.multiplyByVector(new
Vector(theta));

        for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
            vector.increaseComponent(nodesGlobalIndexes[j], localVector.getComponent(j));
        }
    }
```

```
    private void setThirdConditionOnElement(int xStart, int xEnd, int yStart, int yEnd,
                                            double commonMultiplier, Matrix matrix, Vector
vector,
                                            ThreeArityFunction<Double, Double, Double, Double>
function, double time) {
        double[] vectorComponents = new double[EDGE_NODES_COUNT];
        int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

        nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
        nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

        vectorComponents[0] = commonMultiplier
                * function.apply(grid.xValues[xStart], grid.yValues[yStart], time);
        vectorComponents[1] = commonMultiplier
                * function.apply(grid.xValues[xEnd], grid.yValues[yEnd], time);

        Vector localVector = BoundaryConditions.LOCAL_MATRIX.multiplyByVector(new
Vector(vectorComponents));

        for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
            vector.increaseComponent(nodesGlobalIndexes[j], localVector.getComponent(j));
        }

        DenseMatrix localMatrix = new DenseMatrix(BoundaryConditions.LOCAL_MATRIX);
        localMatrix.multiplyByScalar(commonMultiplier);

        for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
            for (int k = 0; k < EDGE_NODES_COUNT; ++k) {
                matrix.increaseComponent(nodesGlobalIndexes[j], nodesGlobalIndexes[k],
localMatrix.getComponent(j, k));
            }
        }
    }

    @Override
    public String toString() {
        return "SolutionArea X: " + Arrays.toString(grid.xValues) + System.lineSeparator()
                + "SolutionArea Y: " + Arrays.toString(grid.yValues);
    }
}
```